ParaSoft®
insure++®

User's Guide

Version 5.2
UNIX

ParaSoft Corporation
2031 S. Myrtle Ave.
Monrovia, CA 91016
Phone: (888) 305-0041
Fax: (626) 305-9048
E-mail: info@parasoft.com
URL: www.parasoft.com

IMPORTANT NOTICE

LIMITED USE OF SOFTWARE LICENSE. This agreement contains the ParaSoft Corporation (PARASOFT) Limited Use Software License Agreement (AGREEMENT) which will govern the use of the ParaSoft products contained within it.

> YOU AGREE TO THE TERMS OF THIS AGREEMENT BY THE ACT OF OPENING THE ENVELOPE CONTAINING THE SOFTWARE OR INSTALLING IT ON YOUR COMPUTER SYSTEM. DO NOT OPEN THE ENVELOPE OR ATTEMPT TO INSTALL THE SOFTWARE WITHOUT FIRST READING, UNDERSTANDING AND AGREEING TO THE TERMS AND CONDITIONS OF THIS AGREEMENT. YOU MAY RETURN THIS PRODUCT TO PARASOFT FOR A FULL REFUND BEFORE OPENING THE ENVELOPE OR INSTALLING THE SOFTWARE.

GRANT OF LICENSE. PARASOFT hereby grants you, and you accept, a limited license to use the enclosed electronic media, user manuals, and any related materials (collectively called the SOFTWARE in this AGREEMENT). You may install the SOFTWARE in only one location on a single disk or in one location on the temporary or permanent replacement of this disk. If you wish to install the SOFTWARE in multiple locations, you must either license an additional copy of the SOFTWARE from PARASOFT or request a multi-user license from PARASOFT. You may not transfer or sub-license, either temporarily or permanently, your right to use the SOFTWARE under this AGREEMENT without the prior written consent of PARASOFT.

DERIVED PRODUCTS. Products developed from the use of the SOFTWARE remain your property. No royalty fees or runtime licenses are required on said products.

TERM. This AGREEMENT is effective from the day you open the sealed package containing the electronic media and continues until you return the original SOFTWARE to PARASOFT, in which case you must also certify in writing that you have destroyed any archival copies you may have recorded on any memory system or magnetic, electronic, or optical media and likewise any copies of the written materials.

PARASOFT’S RIGHTS. You acknowledge that the SOFTWARE is the sole and exclusive property of PARASOFT. By accepting this agreement you do not become the owner of the SOFTWARE, but you do have the right to use the SOFTWARE in accordance with this AGREEMENT. You agree to use your best efforts and all reasonable steps to protect the SOFTWARE from use, reproduction, or distribution, except as authorized by this AGREEMENT. You agree not to disassemble, de-compile or otherwise reverse engineer the SOFTWARE.

YOUR ORIGINAL ELECTRONIC MEDIA/ARCHIVAL COPIES. The electronic media enclosed contain an original PARASOFT label. Use the original electronic media to make “back-up” or “archival” copies for the purpose of running the SOFTWARE program. You should not use the original electronic media in your terminal except to create the archival copy. After recording the archival copies, place the original electronic media in a safe place. Other than these archival copies, you agree that no other copies of the SOFTWARE will be made.

CUSTOMER REGISTRATION. PARASOFT may from time to time revise or update the SOFTWARE. These revisions will be made generally available at PARASOFT’s discretion. Revisions or notification of revisions can only be provided to you if you have signed and returned the enclosed Registration Card to PARASOFT and if your electronic media are the originals. PARASOFT's customer services are available only to registered users.

LIMITED WARRANTY. PARASOFT warrants for a period of thirty (30) days from the date of purchase, that under normal use, the material of the electronic media will not prove defective. If, during the thirty (30) day period, the software media shall prove defective, you may return them to PARASOFT for a replacement without charge. PARASOFT further allows thirty (30) days free consultation regarding the SOFTWARE, its installation and operation. After this initial period PARASOFT reserves the right to refuse further assistance unless a maintenance contract has been purchased.

SUITABILITY. PARASOFT has worked hard to make this a quality product, however PARASOFT makes no warranties as to the suitability, accuracy, or operational characteristics of this SOFTWARE. The SOFTWARE is sold on an "as-is" basis.

EXCLUSIONS. PARASOFT shall have no obligation to support SOFTWARE that is not the then current release.

LIMITATION OF LIABILITY. You agree that PARASOFT's liability for any damages to you or to any other party shall not exceed the license fee paid for the SOFTWARE.

> PARASOFT WILL NOT BE RESPONSIBLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OF THE SOFTWARE ARISING OUT OF ANY BREACH OF THE WARRANTY, EVEN IF PARASOFT HAS BEEN ADVISED OF SUCH DAMAGES. THIS PRODUCT IS SOLD "AS-IS".
>
> SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU. YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

TERMINATION OF AGREEMENT. If any of the terms and conditions of this AGREEMENT are broken, PARASOFT has the right to terminate the AGREEMENT and demand that you return the SOFTWARE to PARASOFT. At that time you must also certify, in writing, that you have not retained any copies of the SOFTWARE.

ENTIRE AGREEMENT. This AGREEMENT constitutes the entire agreement between you and PARASOFT.

All brand and product names are trademarks or registered trademarks of their respective holders.

ParaSoft Corporation
2031 South Myrtle Avenue, Monrovia, CA 91016

Printed in the U.S.A, November 6, 2000

Table of Contents

Insure++ User's Guide

Part I

Insure++ User's Guide

Introduction

Insure++ 5.2 contains significant changes from previous versions that will help you debug faster and more efficiently. These include powerful new technologies and refined user interfaces.

New features/enhancements includes Chaperon for Linux x86, which:

- checks ***all*** data memory references made by a process, whether in the developer's compiled code, language support routines, shared or archive libraries, or operating system kernel calls.
- detects and reports reads of uninitialized memory, reads or writes that are not within the bounds of allocated blocks, and allocation errors such as memory leaks.
- supports multi-threaded programs, including those in Caldera e-Server 2.3 distribution.

If you haven’t already read the *Getting Started* manual, we suggest starting there. If you are interested in a specific subject, you may want to consult the index located at the end of this volume.

Conventions used in this manual

Different typefaces and other symbols will be used in this text to denote various types of information.

```
Text that appears in this typeface is used to
denote source code or the names of functions,
subroutines or variables. It is also used to show
commands that you should type at the keyboard.
```

Offset paragraphs that carry the “dangerous bend” sign are particularly important and should be understood before continuing further.

C++ This symbol in the margin indicates a section specific to C++ users.

TCA This symbol in the margin indicates a section specific to TCA users.

Inuse This symbol in the margin indicates a section specific to Inuse users.

If you get stuck

If you have problems using Insure++, please consult the comprehensive FAQ shipped with your distribution (`FAQ.txt`) first. The FAQ is also kept up-to-date on our web site (the address is on the first page of this manual). You can also obtain a copy of the most recent FAQ by sending an empty email message to `faq@parasoft.com`. If your problem is not described in this document, please follow the procedure below in contacting technical support.

- Check the manual.
- Attempt to isolate a suspected bug to a trivial example. A good method is to remove half of the code and try compiling again, repeating the process until the problem is isolated to ten or twenty lines. Often this procedure can suggest a fix or work-around.
- If the problem is not urgent, report it by e-mail to `support@parasoft.com` or by fax to (626) 305-9048.
- Before calling support at (888) 305-0041, know your Insure++ version. You can easily find it by typing `insure` with no arguments.
- Know your operating system version. You can find it on most Unix platforms by typing `uname -a`.
- If you call, please use a phone near your computer. The support technician may need you to try things while you are on the phone.
- Be prepared to recreate your problem.
- Please be patient.

Thank you for selecting Insure++. Good luck on your journey towards Total Quality Software.

Insure++

As shown in the *Getting Started* manual, using Insure++ is essentially easy to do. You simply recompile your program using the special `insure` command instead of your normal compiler. Running the program normally will then generate a report whenever an error is detected that usually contains enough detail to track down and correct the problem.

What does this give you?

Obviously, the most important advantage of Insure++ is the fact that it automatically detects errors that might otherwise go unnoticed in normal testing. Subtle memory corruption errors and dynamic memory problems often don't crash the program or cause it to give incorrect answers until the program is shipped to customers and they run it on *their* test cases. Then the problems start.

Even if Insure++ doesn't find any problems in your programs, running it gives you the confidence that your program doesn't contain any errors.

Of course, Insure++ can't possibly check everything that your program does. However, its checking is extensive and covers every class of programming error. The following sections discuss the types of errors that Insure++ will detect.

Memory corruption

This is one of the most unpleasant errors that can occur, especially if it is well disguised. As an example of what can happen, consider the program shown in Figure 1, which concatenates the arguments given on the command line and prints the resulting string.

```
/*
 * File: hello.c
 */
int main (int argc, char *arvc[])
{
     int i;
     char str[16];

     str[0] = '\0';
     for(i=0; i<argc; i++) {
          strcat(str, argv[i]);
          if(i < (argc-1)) strcat(str, " ");
     }
     printf("You entered: %s\n", str);
     return (0);
}
```

Figure 1. “Hello world” with bug

If you compile and run this program with your normal compiler, you’ll probably see nothing interesting, e.g.,

```
$ cc -g -o hello hello.c
$ hello
You entered: hello
$ hello world
You entered: hello world
$ hello cruel world
You entered: hello cruel world
```

If this were the extent of your test procedures, you would probably conclude that this program works correctly, despite the fact that it has a very serious memory corruption bug.

If you compile with Insure++, the command “`hello cruel world`” generates the errors shown in Figure 2, because the string that is being concatenated becomes longer than the 16 characters allocated in the declaration at line 7.

```
[hello.c:11] **WRITE_OVERFLOW**
>> strcat(str, argv[i]);

   Writing overflows memory: str

        bbbbbbbbbbbbbbbbbbbbbbbbbbbb
        |          16               |   2   |
        wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww

   Writing (w) : 0xf7fff8a8 thru 0xf7fff8b9
                  (18 bytes)
   To block (b): 0xf7fff8a8 thru 0xf7fff8b7
                  (16 bytes)
             str, declared at hello.c, 7

   Stack trace where the error occurred:
             strcat() (interface)
               main() hello.c, 11

**Memory corrupted. Program may crash!!**

[hello.c:14] **READ_OVERFLOW**
>> printf(“You entered: %s\n”, str);

   String is not null terminated within range: str

   Reading   : 0xf7fff8a8 thru 0xf7fff8b9 (18 bytes)
   From block: 0xf7fff8a8 thru 0xf7fff8b7 (16 bytes)
             str, declared at hello.c, 7

   Stack trace where the error occurred:
               main() hello.c, 16

You entered: hello cruel world
```

Figure 2. Insure++’s messages from the “Hello world” program

Insure++ finds all problems related to overwriting memory or reading past the legal bounds of an object, regardless of whether it is allocated statically (i.e., a global variable), locally on the stack, dynamically (with `malloc` or `new`), or even as a shared memory block.

It also detects the case in which a pointer crosses from one block of memory into another and starts to overwrite memory there, even if the memory blocks are adjacent.

Pointer abuse

Problems with pointers are among the most difficult encountered by C programmers. Insure++ detects pointer related problems in the following categories

- Operations on `NULL` pointers
- Operations on uninitialized pointers
- Operations on pointers that don't actually point to valid data
- Operations that try to compare or otherwise relate pointers that don't point at the same data object
- Function calls through function pointers that don't actually point to functions

Figure 3 shows the code for a second attempt at the "Hello world" program that uses dynamic memory allocation.

The basic idea of this program is that we keep track of the current string size in the variable `length`. As each new argument is processed, we add its length to the `length` variable and allocate a block of memory of the new size. Notice that the code is careful to include the final `NULL` character when computing the string length (line 11) and also the space between strings (line 14). Both of these would be easy mistakes to make. It's an interesting exercise to see how quickly Insure++ would find such an error.

```
/*
 * File: hello2.c
 */
#include <stdlib.h>

int main (int argc, char *argv[]
{
     char *string, *string_so_far;
     int i, length;

     length = 0;    /* Include last NULL */

     for(i=0; i<argc; i++) {
          length += strlen(argv[i])+1;
          string = malloc(length);
/*
 * Copy the string built so far.
 */
          if(string_so_far != (char *)0)
               strcpy(string, string_so_far);
          else *string = '\0';

          strcat(string, argv[i]);
          if(i < argc-1) strcat(string, " ");
          string_so_far = string;
     }
     printf("You entered: %s\n", string);
     return (0);
}
```

Figure 3. "Hello world" with dynamic memory allocation

The code in lines 19-24 either copies the argument to the buffer or appends it depending on whether or not this is the first pass round the loop. Finally in line 25 we point at the new, longer string by assigning the pointer `string` to the variable `string_so_far.`

If you compile and run this program under Insure++, you'll see "uninitialized pointer" errors reported for lines 19 and 20. This is because the variable `string_so_far` hasn't been set to anything before the first trip through the argument loop!

Memory leaks

A "memory leak" occurs when a piece of dynamically allocated memory cannot be freed because the program no longer contains any pointers that point to the block. A simple example of this behavior can be seen by running the (corrected) "Hello world" program with the arguments

```
hello3 this is a test
```

If we examine the state of the program at line 27, just before executing the call to `malloc` for the second time, we observe:

- The variable `string_so_far` points to the string "`hello`", which it was assigned as a result of the previous loop iteration.
- The variable `string` points to the extended string "`hello this`", which was assigned on this loop iteration.

These assignments are shown schematically in Figure 4 - both variables point to blocks of dynamically allocated memory.

The next statement

```
string_so_far = string;
```

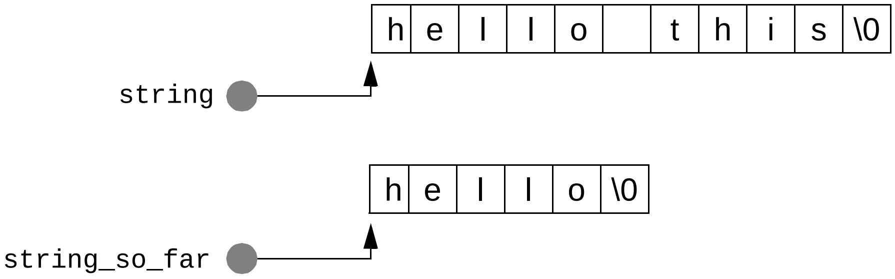


Figure 4. Pointer assignments before the memory leak

will make both variables point to the longer memory block as shown in Figure 5.

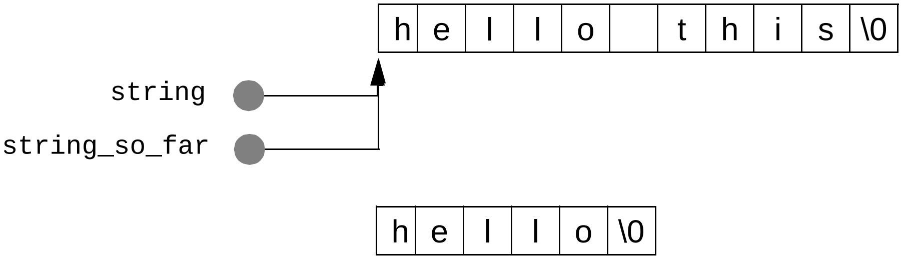
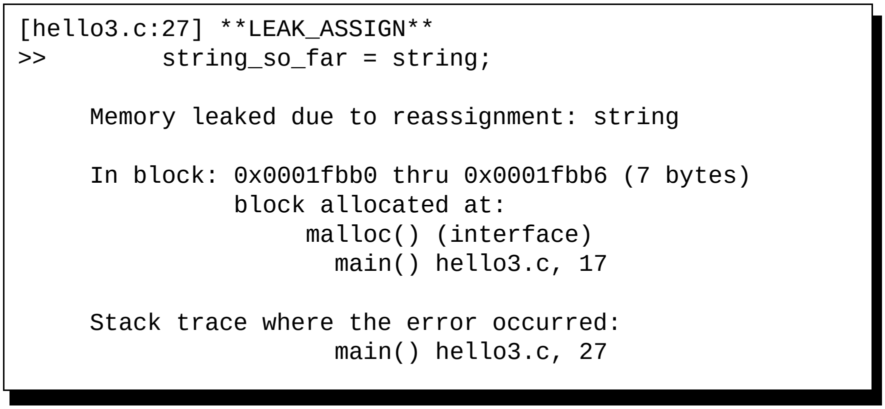


Figure 5. Pointer assignments after the memory leak

Once this has happened, however, there is no remaining pointer that points to the shorter block. Even if you want to, there is no way that you can reclaim the memory that was previously pointed to by `string_so_far`--it is permanently allocated. This is known as a “memory leak”, and is diagnosed by Insure++ as shown in Figure 6.



```
[hello3.c:27] **LEAK_ASSIGN**
>>        string_so_far = string;

     Memory leaked due to reassignment: string

     In block: 0x0001fbb0 thru 0x0001fbb6 (7 bytes)
               block allocated at:
                    malloc() (interface)
                      main() hello3.c, 17

     Stack trace where the error occurred:
                      main() hello3.c, 27
```

Figure 6. Insure++ error report for the memory leak

This example is called `LEAK_ASSIGN` by Insure++ since it is caused when a pointer is re-assigned. Other types of leaks that Insure++ detects include:

`LEAK_FREE` Occurs when you free a block of memory that contains pointers to other memory blocks. If there are no other pointers that point to these secondary blocks then they are permanently lost and will be reported by Insure++.

`LEAK_RETURN` Occurs when a function returns a pointer to an allocated block of memory, but the returned value is ignored in the calling routine.

`LEAK_SCOPE` Occurs when a function contains a local variable that points to a block of memory, but the function returns without saving the pointer in a global variable or passing it back to its caller.

Notice that Insure++ indicates the exact source line on which the problem occurs, which is a key issue in finding and fixing memory leaks. This is an extremely important feature, because it is easy to introduce subtle memory leaks into your applications but very hard to find them all. Using Insure++, you can instantly pinpoint the line of source code which caused the leak.

Should memory leaks be fixed?

Whether or not this is a serious problem depends on your application. To get more information on the seriousness of the problem, make a file called `.psrc` in your current directory and add to it the line[1]

```
insure++.summarize leaks
```

Now when you run the program again, you will see the same output as before, followed by a summary of all the memory leaks in your code.

```
MEMORY LEAK SUMMARY
===================

4 outstanding memory references for 55 bytes.

Leaks detected during execution
-------------------------------
      55 bytes 4 chunks allocated at hello3.c, 17
```

This shows that even this short program lost four different chunks of memory. The total of 55 bytes isn't very large and you might well ignore it in a program this size. If, however, this was a routine in a larger program, it would be a serious problem, because every time the routine is called it allocates blocks of memory and loses some. As a result the program gradually consumes more and more memory and will finally crash when the memory space on the host machine is exhausted.

[1] If you already have a file called `.psrc` in your directory, simply add this line to it.

This type of bug can be extremely hard to detect, because it might take literally days to show up. It is exactly the type of bug that survives all your in-house testing and only shows up when you ship a product to a customer who needs to use it for some enormous processing task!

You may be wondering why Insure++ only prints one error message although the summary indicates that 4 memory leaks occurred. This is because Insure++ normally shows only the first error of any given type at each particular source line. If you wish, you can change this behavior as described in “Insure++ Reports” on page 59.

You can obtain additional information about each individual memory leak with the `.psrc` option “insure++`.summarize leaks`” (see page 75 and page 163).

Finding *all* memory leaks

Inuse

For an even higher level of checking, we suggest the following algorithm for removing all memory leaks from your code. This process is unique - no other tool can do this. If you complete the following steps, there will not be any memory leaks left in your code.

1. Compile your program normally, but link with `insure -Zuse` and run the program with Inuse (see “Using Inuse” in the Inuse manual). If you see an increase in the heap size as you run the program, you are leaking memory.
2. Compile all source code, but not libraries, with Insure++. Clean all leaks that are detected by Insure++.
3. Compile everything that makes up your application with Insure++--source code and libraries. Clean any leaks detected by Insure++. If you do not have source for any of the libraries, skip this step and proceed to Step 4.

4. Repeat Step 1. If memory is increasing, add `insure++.summarize leaks outstanding` to your `.psrc` file and run your Insure++-checked program again. Any outstanding memory reference shown is a potential leak.
5. You must now examine each outstanding memory reference to determine whether it is a leak. If the pointer is passed into a library function, it may be saved. If this is the case, it is not a leak. Once every outstanding memory reference is understood, and those that are leaks are cleared, the program is free of memory leaks.

Dynamic memory manipulation

Using dynamically allocated memory properly is another tricky issue. In many cases programs continue running well after a programming error causes serious memory corruption; sometimes they don’t crash at all.

One common mistake is to try to reuse a pointer after it has already been freed.

As an example we could modify the “Hello world” program to de-allocate memory blocks before allocating the larger ones. Consider the following piece of code which does just that:

```
21:   if(string_so_far != (char *)0) {
22:       free(string_so_far);
23:       strcpy(string, string_so_far);
24:   }
25:   else *string = ’\0’;
```

If you run this code (`hello4.c`) through Insure++, you’ll get another error message about a “dangling pointer” at line 23. The term “dangling pointer” is used to mean a pointer that doesn’t point at a valid memory block anymore. In this case the block is freed at line 22 and then used in the following line!

This is another common problem that often goes unnoticed, because many machines and compilers allow this particular behavior.

In addition to this error Insure++, also detects the following:

- Reading from or writing to “dangling pointers”
- Passing “dangling pointers” as arguments to functions or returning them from functions
- Freeing the same memory block multiple times
- Attempting to free statically allocated memory.
- Freeing stack memory (local variables)
- Passing a pointer to `free` that doesn’t point to the beginning of a memory block
- Calls to `free` with `NULL` or uninitialized pointers
- Passing nonsensical arguments or arguments of the wrong data type to `malloc`, `calloc`, `realloc` or `free`

Another way that Insure++ can help you track down dynamic memory problems is through the `RETURN_FAILURE` error code. Normally, Insure++ will not issue an error if `malloc`, for example, returns a `NULL` pointer because it is out of memory. This behavior is the default, because it is assumed that the user program is already checking for, and handling, this case.

If your program appears to be failing due to an unchecked return code, you can enable the `RETURN_FAILURE` error message class (See page 342). Insure++ will then print a message whenever any system call fails.

Strings

The standard C library string handling functions are a rich source of potential errors, since they do very little checking on the bounds of the objects being manipulated.

Insure++ detects problems such as overwriting the end of a buffer as described in “Memory corruption” on page 5. Another common problem is caused by trying to work with strings that are not null-terminated, as in the following example.

```
1:    /*
2:     * File: readovr2.c
3:     */
4:    int main()
5:    {
6:         char junk;
7:         char b[8];
8:         strncpy(b, “This is a test”,
9:                              sizeof(b));
10:        printf("%s\n", b);
11:        return (0);
12:   }
```

This program attempts to copy the string “`This is a test`” into a buffer which is only 8 characters long. Although it uses `strncpy` to avoid overwriting its buffer, the resulting copy doesn’t have a `NULL` on the end. Insure++ detects this problem in line 10 when the call to `printf` tries to print the string.

Uninitialized memory

A particularly unpleasant problem to track down occurs when your program makes use of an uninitialized variable. These problems are often intermittent and can be particularly difficult to find using conventional means, since any alteration in the operation of the program may result in different behavior. It is not unusual for this type of bug to show up and then immediately disappear whenever you do something to try to trace it.

Insure++ performs checking for uninitialized data in two sub-categories

`copy` Normally, Insure++ doesn’t complain when you assign a variable using an uninitialized value, since many applications do this without error. In many cases the value is changed to something correct before being used, or may never be used at all.

`read` Insure++ generates an error report whenever you use an uninitialized variable in a context which cannot be correct, such as an expression evaluation.

To clarify the difference between these categories consider the following code

```
1:    /*
2:     * File: readuni1.c
3:     */
4:    #include <stdio.h>
5:
6:    int main()
7:    {
8:        struct rectangle {
9:            int width;
10:           int height;
11:       };
12:
13:       struct rectangle box;
14:       int area;
15:
16:       box.width = 5;
17:       area = box.width*box.height;
18:       printf("area = %d\n", area);
19:       return (0);
20:   }
```

In line 17 the value of `box.height` is used to calculate a value that is most definitely invalid, since its value was never assigned. Insure++ detects this error in the `READ_UNINIT_MEM(read)`category. This category is enabled by default, so a message will be displayed.

If you changed line 17 to

```
17:       area = box.height;
```

Insure++ would report errors of type READ_UNINIT_MEM(copy) for both lines 17 and 18, but only if you had unsuppressed this error category.

Uninitialized memory detection options

In a significant change from earlier versions, Insure++ now detects uninitialized memory references using a full flow-analysis of your application's source code (and can often detect problems at compile time) by default. In addition to the performance enhancements made to enable this change, there are several new .psrc options, which allow greater control over this portion of Insure++'s checking abilities (see page 154).

The default setting is the most comprehensive form of error detection, but obviously involves some overhead during compilation. If you wish to track only uninitialized pointers, you can set the following .psrc option.

```
insure++.checking_uninit off
```

Turning off this option does not, however, completely disable uninitialized variable checking. No errors will be reported in the READ_UNINIT_MEM class, but Insure++ will still check for uninitialized pointer variables and report these errors in the READ_UNINIT_PTR error category.

If checking_uninit is disabled, uninitialized pointer errors will be reported in the READ_UNINIT_PTR category, not READ_UNINIT_MEM.

Unused variables

Insure++ can also detect variables that have no effect on the behavior of your application, either because they are never used, or because they are assigned values that are never used. In most cases these are not serious errors, since the offending statements can simply be removed, and so they are suppressed by default.

Occasionally, however, an unused variable may be a symptom of a logical program error, so you may wish to enable this checking periodically. See “Unused variables” on page 347 for more details.

Data representation problems

A lot of programs make either explicit or implicit assumptions about the various data types on which they operate. A common assumption made on workstations is that pointers and integers have the same number of bytes. While some of these problems can be detected during compilation, some codes go to great lengths to hide operations with typecasts such as

```
char *p;
int ip;

ip = (int)p;
```

On many systems this type of operation would be valid and would cause no problems. When such code is ported to alternative architectures, however, problems can arise. The code shown above would fail, for example, when executed on a PC (16-bit integer, 32-bit pointer) or a 64-bit architecture such as the DEC Alpha (32-bit integer, 64-bit pointer).

In cases where such an operation loses information, Insure++ will report an error. On machines for which the data types have the same number of bits (or more), no error is reported.

Incompatible variable declarations

Insure++ detects inconsistent declarations of variables between source files.

A common problem is caused when an object is declared as an array in one file, e.g.

```
int myblock[128];
```

but as a pointer in another

```
extern int *myblock;
```

See the files `baddecl1.c` and `baddecl2.c` for an example. Insure++ also reports differences in size, so that an array declared as one size in one file and another in a second will be detected.

I/O statements

The `printf` and `scanf` family of functions are easy places to make mistakes which show up either as bugs or portability problems.

Consider, for example, the code

```
foo()
{
      double f;

      scanf("%f", &f);
}
```

This code will not crash, but the value read into the variable `f` will not be correct, since its data type (`double`) doesn't match the format specified in the call to `scanf` (`float`). As a result, incorrect data will be transferred to the program.

In a similar way, the example `badform2.c`

```
foo()
{
      float f;

      scanf("%lf", &f);
}
```

corrupts memory, since too much data will be written over the supplied variable. This error can be very difficult to detect.

Insure++ detects both of these bugs.

A more subtle issue arises when data types used in I/O statements match "accidentally". The code

```
foo()
{
      long l = 123;
      printf("l = %d\n", l);
}
```

functions correctly on machines where types `int` and `long` have the same number of bits, but fails otherwise. Insure++ detects this error, but classifies it differently from the previous cases. You can choose to ignore this type of problem while still seeing the previous bugs. (See page 190 for details.)

In addition to checking `printf` and `scanf` arguments, Insure++ also detects errors in other I/O statements. The code

```
foo(line)
      char line[80];
{
        gets(line);
}
```

works as long as the input supplied by the user is shorter than 80 characters, but fails on longer input. Insure++ checks for this case and reports an error if necessary.

This case is somewhat tricky, since Insure++ can only check for an overflow after the data has been read. In extreme cases the act of reading the data will crash the program before Insure++ gets the chance to report it.

Mismatched arguments

Calling functions with incorrect arguments is a common problem in many programs, and can often go unnoticed.

Insure++ detects the error in the following program

```
double foo(dd)
        double dd;
{
        return dd + 1.0;
}

main()
{
        printf("Result = %f\n", foo(1));
}
```

in which the argument passed to the function `foo` in `main` is an integer rather than a floating point number.

Converting this program to ANSI style (e.g., with a function prototype for `foo`) makes it correct since the argument passed in `main` will be automatically converted to `double`. Insure++ doesn't report an error in this case.

Insure++ detects several different categories of errors, which you can enable or suppress separately depending on which types of bugs you consider important.

Sign errors — Arguments agree in type but one is signed and the other unsigned, e.g., `int` vs. `unsigned int`.

Compatible types — The arguments are different data types which happen to occupy the same amount of memory on the current machine, e.g. `int` vs. `long` if both are 32 bits. While this error may not cause problems on your current machine, it is a portability problem.

Incompatible types — Similar to the example above - data types are fundamentally different or require different amounts of memory. `int` vs. `long` would appear in this category on machines where they require different numbers of bits.

C++ compile time warnings

During compilation, Insure++'s parser detects a number of C++-specific problems and prints warning messages. These messages are coded by the chapter, section, and paragraphs pertaining to that warning in the draft ANSI standard. Therefore, if you are uncertain what a particular warning message means or would like additional information, you can consult the standard for an explanation.

As an example, when processed by Insure++, the code

```
void foo(char *str) { }
void func()
{
      void *iptr = (char *) 0;
      foo(iptr);
}
```

will produce the warning

```
insure -c foo.C
[foo.C:5] Warning:13-2: wrong arguments passed to
function ’foo’
|   declared at: [foo.C:1]
| expected args: (char *)
|   passed args: (void *)
>> foo(iptr);
```

Invalid parameters in system calls

Interfacing to library software is often tricky, because passing an incorrect argument to a routine may cause it to fail in an unpredictable manner. Debugging such problems is much harder than correcting your own code, since you typically have much less information about how the library routine should work.

Insure++ has built-in knowledge of a large number of system calls and checks the arguments you pass to ensure correct data type and, if appropriate, correct range.

For example, the code

```
void myrewind(FILE fp)
{
      fseek(fp, (long)0, 3);
}
```

would generate an error since the last argument passed to the `fseek` function is outside the legal range.

Unexpected errors in system calls

Checking the return codes from system calls and dealing correctly with all the error cases that can arise is a very difficult task. It is a very rare program that deals with all possible cases correctly.

An unfortunate consequence of unusual error cases is that programs can fail unexpectedly after they have been shipped to customers because some system call fails in a way that had not been anticipated. The consequences of this failure can range from a nasty “core dump” to a system that performs erratically at the customer location.

Insure++ has a special error class, `RETURN_FAILURE`, that can be used to detect these problems. All the system calls known to Insure++ contain special error checking code that detects failures. Normally these errors are suppressed, since it is assumed that the application is handling them itself, but they can be enabled at runtime by adding the line

```
insure++.unsuppress RETURN_FAILURE
```

to a `.psrc` file. Any system call that returns an error code will then print a message indicating the name of the routine, the arguments supplied, and the reason for the error.

This capability detects *any* error in *any* known system call. Among the potential benefits are automatic detection of errors in the following situations:

- `malloc` runs out of memory
- Files that don’t exist
- Incorrectly set permission flags
- Incorrect use of I/O routines
- Exceeding the limit on open files
- Inter-process communication and shared memory errors
- Unexpected “interrupted system call” errors

and many others.

Threads

In order for insure runtime to be able to track memory correctly in threaded programs, all calls to `pthread_create()` or `thr_create()` must have been "seen" by Insure++. In Insure++ 3.1 and earlier versions this meant that all files that ever call thread creation routines must be instrumented with Insure++. This was still true in version 4.0 if you were using backward compatibility mode (TQI interfaces). Starting with version 4.0, Insure++ has defaulted to using "Library Interpositioning" (referred to as TQL interfaces from now on). This mode guarantees that the above requirement is met, even if Insure++ was only used to link the executable, and none of the source files have been instrumented.

Achieving Total Quality Software

The previous sections described the various types of problems detected by Insure++. As you can see, you can detect a very large number of problems by simply recompiling your program and running it under Insure++. Running under Insure++ will eliminate many bugs that you might otherwise ship to your customers.

It would be naive, however, to expect that Insure++ will remove all of the bugs in your code. Some will still make it through all the testing steps. Luckily, Insure++ can still help even after you’ve shipped your product.

An important way that Insure++ can help you reach the Total Quality Software goal is to ship two versions of your product to your customers:

- The normal version, compiled without Insure++
- A version built with Insure++

This second version can be used *at the customer site* to help track down problems. This will dramatically improve the efficiency of your support staff at finding bugs in the released software.

Selective Checking

By default, Insure++ will check for bugs for the entire duration of your program. If you are only interested in a portion of your code, you can make some simple, unobtrusive changes to the original source to achieve this.

When you compile with `insure`, the pre-processor symbol `__INSURE__` is automatically defined. This allows you to conditionally insert calls to enable and disable runtime checks.

Suppose, for example, that you are not interested in events occurring during the execution of a hypothetical function `grind_away`. To disable checking during this function, you can modify the code as shown below

```
grind_away() {
#ifdef __INSURE__
      _Insure_set_option("runtime", "off");
#endif
      ... code ...
#ifdef __INSURE__
      _Insure_set_option("runtime", "on");
#endif
}
```

Now when you compile and run your program, it will not check for bugs between the calls to `_Insure_set_option`.

Alternatively, if you do not want to modify the code for the `grind_away` function itself, you can add calls to `_Insure_set_option` around the calls to `grind_away`.

Interacting with Debuggers

While it is our intent that the error messages generated by Insure++ be sufficient to identify most programming problems, it will sometimes be useful to have direct access to the information known to Insure++. This can be useful in the following situations:

- You are running your program from a debugger and would like to cause a breakpoint whenever Insure++ discovers a problem.
- You are tracing an error using the debugger and would like to monitor what Insure++ knows about your code.
- You wish to add calls to your program to periodically check the status of some data.

Available functions

Whenever Insure++ detects an error, it prints a diagnostic message and then calls the routine `_Insure_trap_error`. This is a good place to insert a breakpoint if you are working with a debugger.

The following functions show the current status of memory and can be called either from your program or the debugger. Remember to add prototypes for the functions you use, particularly if you are calling these C functions from C++ code.

```
int _Insure_mem_info(void *pmem);
```

Displays information that is known about the block of memory at address `pmem`. (Returns zero.)

```
int _Insure_ptr_info(void **pptr);
```

Displays information about the pointer at the indicated address. (Returns zero.)

The following function lists all currently allocated memory blocks, including the line number at which they were allocated. It can be called directly from your program or from the debugger.

```
long _Insure_list_allocated_memory(int mode);
```

0 - Just the total allocation
1 - “Newly-Allocated” or reallocated blocks
2 - Everything

Sample debugging session

The use of these functions is best illustrated by example.

Consider the following program

```
/*
 * File: bugsfunc.c
 */
#include <stdlib.h>

main()
{
    char *p, *q;

    p = (char *)malloc(100);

    q = "testing";
    while(*q) *p++ = *q++;

    free(p);
}
```

Compile this code under Insure++ in the normal manner (with the `-g` option), and start the debugger in the normal manner.

The instructions shown here assume that the debugger you are using is similar to `dbx`. If you are using another debugger, similar commands should be available.

```
$ dbx bugsfunc
Reading symbolic information...
Read 4650 symbols
(dbx)
```

If the debugger has trouble recognizing and reading the source file, you may need to use the `rename_files on .psrc` option. See page 150 for more information about this option.

It is generally useful to put a breakpoint in `_Insure_trap_error` so that you can get control of the program whenever an error occurs. In this case, we run the program to the error location with the following result

```
(dbx) stop in _Insure_trap_error
(2) stop in _Insure_trap_error
```

The above may not work if you have linked against the shared Insure++ libraries (the default). If you cannot set a breakpoint as shown above, it is because the shared libraries are not loaded by the debugger until the program begins to run. You can avoid this problem by linking against the static Insure++ libraries (see the `static_linking on` option on page 151) or by setting a breakpoint in `main` and starting the program before setting the breakpoint on `_Insure_trap_error`.

```
(dbx) run
Running: bugsfunc
[bugsfunc.c:15] **FREE_BODY**
>>        free(p);

Freeing memory block from body: p

Pointer : 0x0001e26f
Stack trace where the error occurred:
          main() bugsfunc.c, 15

**Memory corrupted. Program may crash!!**
stopped in _Insure_trap_error at 0xef6d8620
0xef6d8620: _Insure_trap_error: save %sp,-OxTc%sp
(dbx)
```

The program is attempting to free a block of memory by passing a pointer that doesn’t indicate the start of an allocated block. The error message shown by Insure++ identifies the location at which the block was allocated and also shows us that the variable `p` has been changed to point into the middle of the block, but it doesn’t tell us where the value of `p` changed.

We can use the Insure++ functions from the debugger to help track this down.

Since the program is already in the debugger, we can simply add a breakpoint back in `main` and restart it

```
(dbx) func main
(dbx) stop at 10
(4) stop at "examples/bugsfunc.c":10
(dbx) run
Running: bugsfunc
stopped in main at line 10 in file "bugsfunc.c"
      10  p = (char *)malloc(100);
```

To see what is currently known about the pointers p and q, we can use the _Insure_ptr_info function

Note that the _Insure_ptr_info function expects to be passed the *address* of the pointer, not the pointer itself. To see the contents of the memory indicated by the pointers, use the _Insure_mem_info function.

```
(dbx) print _Insure_ptr_info(&p)
  Uninitialized
(dbx) print _Insure_ptr_info(&q)
  Uninitialized
```

Both pointers are currently uninitialized, as would be expected.

To see something more interesting, we can continue to line 13 and repeat the previous steps.

```
(dbx) stop at 13
(6) stop at "examples/bugsfunc.c":13
(dbx) cont
stopped in main at line 13 in file "bugsfunc.c"
      13   while(*q) *p++ = *q++;
(dbx) print _Insure_ptr_info(&p)
  Pointer  : 0x0001e358 (heap)
  Offset   : 0 bytes
  In Block : 0x0001e358 thru 0x0001e3bb
             (100 bytes)
             p, allocated at bugsfunc.c, 10
```

The variable p now points to a block of allocated memory. You can check on all allocated memory by calling _Insure_list_allocated_memory.

```
(dbx) print _Insure_list_allocated_memory()
1 allocated memory block, occupying 100 bytes.
[bugsfunc.c:10] 100 bytes.
```

Finally, we check on the second pointer, q.

```
(dbx) print _Insure_ptr_info(&q)
  Pointer  : 0x00016220 (global)
  Offset   : 0 bytes
  In Block : 0x00016220 thru 0x0001e3bb
             (100 bytes)
             "testing", declared at bugsfunc.c,12
```

Everything seems OK at this point, so we can continue to the point at which the memory is freed and check again.

```
(dbx) next
      15   free(p);
(dbx) print _Insure_ptr_info(&p)
  Pointer  : 0x0001e35f (heap)
  Offset   : 7 bytes
  In Block : 0x0001e358 thru 0x0001e3bb
             (100 bytes)
             p, allocated at bugsfunc.c, 10
```

The critical information here is that the pointer now points to an offset 7 bytes from the beginning of the allocated block. Executing the next statement, free(p), will now cause the previously shown error, since the pointer doesn't point to the beginning of the allocated block anymore.

Since everything was correct at line 12 and is now broken at line 15, it is simple to find the problem in line 13 in which the pointer p is incremented while looping over q.

Tracing

Tracing is a very useful enhancement of Insure++ for C++ programmers. Because C++ is such a complicated language, programmers may never know which functions are being called or in which order. Some functions are called during initialization before the main program begins execution. Tracing provides the programmer with the ability to see how functions, constructors, destructors, and more are called as the program runs.

Insure++ prints a message at the entry to every function which includes the function name, filename, and line number of the command that called it.

A typical line of output from tracing looks like:

```
function_name filename, line_number
```

By default, the output is indented to show the proper depth of the trace.

Turning tracing on

By default, tracing is turned off. The easiest way to turn tracing on is to use the `trace on` option in your `.psrc` file. This turns on tracing for the entire program. See page 164 for more information about this option.

To get a full trace, you must use the `-g` compiler switch on your `insure` compile line. To get file names and line numbers in the trace output, you must use the `stack_internal on` option when compiling your program. (see page 151)

You may not want to do this always, though, because your program will slow down while every function call prints information.

This problem can be minimized by selectively turning on tracing during the execution of your program only in those sections of the code where you need it most. This can be done using the special Insure++ command

```
void _Insure_trace_enable(int flag)
```

`flag` = 0 turns tracing off
`flag` = 1 turns tracing on

There is one more special Insure++ function that works with tracing. This function may be used to add your own messages to the trace.

```
void _Insure_trace_annotate(int indent,
                            char *format, ...)
```

`indent` = 0 means string is placed in column zero
`indent` = 1 means string will be indented at proper level
`format` should be a normal printf-style format string

Directing tracing output to a file

The default output for tracing data, like all other Insure++ output, is `stderr`. You can direct the output to a file using the `trace_file filename` option in your `.psrc` file. See page 165 for more information about this option.

When you use this option, Insure++ prints a message reminding you where the tracing data is being written. If you would like to eliminate these reminders, you can use the `trace_banner off` option in your `.psrc` file. See page 164 for more information.

Example

Consider the following code, which will illustrate how tracing works.

```
1:   /*
2:    * File: trace.C
3:    */
4:   int twice(int j) {
5:        return j*2;
6:   }
7:   class Object {
8:   public:
9:        int i;
10:       Object() {
11:            i = 0;
12:       }
13:       Object(int j) {
14:            i = j;
15:       }
16:       operator int() { return twice(i); }
17:  };
18:  int main() {
19:       Object o;
20:       int i;
21:
22:       i = o;
23:       return i;
24:  }
```

If you compile and link `trace.C` with the `-g` option and the `stack_internal on` option (see page 151), and then run the executable with the `trace on` option in your `.psrc` file, you will see the following output in `stderr`.

```
main
  Object::Object                         trace.C, 19
  Object::operator int                   trace.C, 22
    twice                                trace.C, 16
```

Signals

In addition to its other error checks, Insure++ also traps certain signals. It does this by installing handlers when your program starts up. These do not interfere with your program's own use of signals; any code that manipulates signals will simply override the functions installed by Insure++.

Signal handling actions

When a signal is detected, Insure++ does the following:

- Prints an informative error
- Logs the signal in the Insure++ report file, if one is being used
- Calls the function `_Insure_trap_error`
- Takes the appropriate action for the signal

If this last step will result in the program terminating, Insure++ attempts to close any open files properly. In particular, the Insure++ report file will be closed. Note that this can only work if the program hasn't crashed the I/O system. If, for example, the program has generated a "bus" or similar error, it might not be possible to close the open files. In the worst of all possible scenarios you will simply generate another (fatal) signal when Insure++ attempts to clean up.

The third step is useful if you are working with a debugger, as described in "Interacting with Debuggers" on page 31. In this case, you can insert a breakpoint at `_Insure_trap_error` and have the program stop whenever it is generating one of the trapped signals.

Interrupting long-running jobs

Insure++ installs a handler for the keyboard interrupt command (often CTRL-C, or the delete key). If your program does not override this handler with one of its own, you can abort a long-running program and still get Insure++'s output. If your program has its own handler for this sequence, you can achieve the same effect by adding the following lines to your handler

```
#ifdef _INSURE_
      _Insure_cleanup();
#endif
```

Which signals are trapped?

By default, Insure++ traps the following signals

```
SIGABRT
SIGBUS
SIGEMT
SIGFPE
SIGILL
SIGINT
SIGIOT
SIGQUIT
SIGSEGV
SIGSYS
SIGTERM
SIGTRAP
```

You can add to or subtract from this list, by adding lines to one of your `.psrc` files and re-running the program.

Signals are added to the list with options such as

```
        insure++.signal_catch SIGSTOP SIGCLD SIGIO
```

and removed with

```
        insure++.signal_ignore SIGINT SIGQUIT SIGTERM
```

You can omit the “`SIG`” prefix if you wish.

Chaperon (Linux Only)

Introduction

Application developers using Linux x86 can now ensure the good behavior of their application by taking advantage of comprehensive memory checking. Chaperon checks all data memory references made by a process, whether in the developer's compiled code, language support routines, shared or archive libraries, or operating system kernel calls. Chaperon detects and reports reads of uninitialized memory, reads or writes that are not within the bounds of allocated blocks, and allocation errors such as memory leaks.

Chaperon works with existing executable programs. In most cases, Chaperon requires no recompilation and no relinking, and no changes to environment variables. Just add `Chaperon` to the beginning of the command line; Chaperon will run the process and check all data memory references.

When Chaperon detects improper behavior, it issues an error message identifying the kind of error and where it occurred. Improper behavior is any access to a logically unallocated region, or a Read (or Modify) access to bytes which have been allocated but not yet Written. The Read fetched uninitialized bytes from memory, and the resulting value is logically undefined. Improper behavior also includes errors or abuses of the `malloc`/`free` protocol, such as attempting to free the same block twice.

Chaperon also detects memory blocks that have been allocated and not freed. Such a block is "in use." If a block is in use and unreachable by starting from the stack, or statically allocated regions, and proceeding through already reached allocated blocks, then the block is a "memory leak." The block could not be freed without some oracle to specify its address as the parameter to `free()`. At `exit()` Chaperon reports memory leaks automatically, and reports blocks in use if requested by the command line option:

```
Chaperon %program_name%
```

Using Chaperon does not require running under a debugger, but Chaperon also works with existing debuggers such as gdb. For more information, see “Using Chaperon With gdb” on page 56.

Requirements and limitations

Requirements

- ELF format executables and shared libraries, with `/lib/ld-linux.so.2 -> ld-2.1.1.so` (or compatible), as the ELF program interpreter. The important interfaces are `_dl_runtime_resolve`, `_dl_relocate_object _dl_debug_state`, and `_r_debug`.
- Any x86 processor [x >= 3] running Linux. In case of opcode conflict between manufacturers, Chaperon follows the Intel documentation.
- Linux kernel 2.2.5 or compatible. Other kernels will work with adjustment of the accounting for system calls.
- No kernel threads. There must be only one actual execution context per process, and Chaperon must see all the switching between register sets.
- `malloc`/`free`/etc must not call `sigaction` that gets used.
- `vfork()` is remapped to `fork()`. Programs depending on `vfork()` semantics may not work properly.
- 32-bit code (no `0x67` address size prefix; but `0x66` operand size prefix is OK), flat model. Any explicit `cs`, `ds`, or `ss` segment selector in the instruction stream must equal the corresponding current actual selector. Chaperon's access accounting treats all offsets as belonging to segment `ds`.

Application code using `es`, `fs`, or `gs` does run; but the accounting may become confused.

Bitfields

Chaperon accounts for memory on a byte-by-byte basis. Since the mapping between bytes and bitfields need not be 1-to-1 and onto, there are problems. By default, Chaperon uses heuristics to guess that some instruction sequences (that would otherwise generate complaints of Read before Write) correspond to legitimate bitfield operations, and the heuristics enable Chaperon to suppress those complaints. They also cause a whole byte to be marked as Written as soon as the first write to any bitfield that intersects it. The heuristics are not complete; there will still be "false positive" complaints of Read before Write. Some source statements and expressions that are not bitfields can generate code that looks like bitfields, and for which the heuristics should be disabled; use the command line parameter `-bitfields=0`. The general palliative for cleaning up Chaperon complaints about bitfields is to clear all words that contain bitfields as soon as the memory is allocated, perhaps using `memset` and perhaps employing a `union`. This can even be more efficient, but some programmers consider it to be distasteful.

Symbols, tracebacks, and compilers

Chaperon’s underlying execution engine and tracking for memory state depend only on x86 architecture, and are compiler independent. But the generation and reporting of tracebacks and symbols relies on Chaperon being able to find and identify the code and symbols. Chaperon recognizes the functions involved in dynamic binding (`.dynsym` symbols), static binding (`.symtab` symbols), and one flavor of debugging (`.stab` symbols). Chaperon also handles the anonymous functions it finds by taking the transitive closure of the static lexical call graph, and their immediate successors in the `.text` section. (In most cases, Chaperon correctly separates and identifies anonymous functions that `gdb` lumps together with the preceding named function.) For modules that have been stripped of symbols, this can omit functions that are called only indirectly through a pointer or table, perhaps including some C++ virtual functions. The remedy is not to strip such modules of their symbols.

The `jmp` table code generated for `switch` statements also can be problematic for Chaperon to understand, and may require updates to accommodate different compilers.

System calls

Chaperon checks the documented memory access behavior of kernel calls for Linux 2.2.x except `bdflush`, `capget`, `capset`, `getpmsg`, `ipc` (but `shm*` shared memory calls *are* checked), `modify_ldt`, `nfsservctl`, `prctl`, `putpmsg`, `quotactl`, `sysfs`, and `vm86*`. Chaperon understands the "regular" `SYS_ioctl` calls whose command word uses `_IOR`, `_IOW`, or `_IOWR`, plus important non-regular cases such as `TIOC*` (terminal control) and `SIOC*` (socket control).

Space

Chaperon runs in the same execution context and address space as the process that Chaperon is checking. The linear coefficients of space overhead are 2 bits of accounting info per byte of address space used by the application, plus (`16 + 8*traceback_length`) bytes per active allocated block. Process sizes greater than about 500MB have not been well explored.

Memory states and access accounting

State	Read or Modify Access	Write Access
Unallocated	error: Read before Allocate	error: Write before Allocate

Allocated but not Written	error: Read before Write	OK; becomes Allocated and Written
Allocated and Written	OK	OK

See also “Bitfields” on page 47.

Allocators: `malloc`, `calloc`, `realloc`, `memalign`, `__libc_malloc`, `__libc_calloc`, `__libc_realloc`, `__libc_memalign`, stack growth (push, create frame), `__brk`, `brk`, `__sbrk`, `sbrk`, `mmap`

De-allocators: `free`, `realloc`, `__libc_free`, `__libc_realloc`, `__brk`, `brk`, `__sbrk`, `sbrk`, stack trim (pop, delete frame), `munmap`

Other known functions: `memcpy`, `memset`, `memmove`, `memchr`, `bcopy`, `bzero`, `strcat`, `strchr`, `__stpcpy`, `strcpy`, `strrchr`. These are optimized for faster performance, and/or to reduce the clutter of multiple error messages that arise from a single call, and/or to suppress “false positive” Read before Write messages from instruction sequences that are known to be used to implement write-allocate cache control, or speculative word-wide reading of byte arrays.

Handling of `realloc(ptr, size)`:

```
If 0==size then free(ptr);
else if 0==ptr then mlloc(size);
else {free(ptr); malloc(size)}
```

and arrange for the new contents of the `malloc()`ed region to equal the old contents for the first min(old_size, new_size) bytes.

Examples

Note: The numeric values of addresses might not match when the examples are re-run. For instance, locations in the stack (`0xbfffffff` and lesser nearby locations) depend on the number of characters in environment variables. Locations in shared libraries (`0x40000000` and greater nearby locations) change with different versions and different values of `LD_PRELOAD`. Locations in application code (`0x08040000` and greater nearby locations) depend on compiler and compiler options.

WRITE_OVERFLOW

To run this exampe, first, compile the code with gcc.

```
gcc -g -o writover writover.c
```

Run the new executable on Chaperon.

```
Chaperon writover
```

Chaperon should report errors such as:

```
#----- writover.c
#include <stdlib.h>

int
main()
{
        /* An example of Write before Allocate */
        char *p = malloc(10);
        p[11] = 3;
        return 0;
}
#-----
$ $HOME/bin/Chaperon ./writover
        # Chaperon banner and license info
// Chaperon(tm) memory access checker version 2.0  2000-07-07.
// Copyright 1999 BitWagon Software LLC.  All rights reserved.
// Copyright 2000 ParaSoft Corp.          All rights reserved.
[writover.c:8] **WRITE_OVERFLOW**
>>         p[11] = 3;

  Writing overflows memory.

          bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
          |              10               | 1 | 1 |
                                             wwwww

   Writing  (w) : 0x0804958b thru 0x0804958b (1 byte)
   To block (b) : 0x08049580 thru 0x08049589 (10 bytes)
                 block allocated at writover.c, 7
                            main()  writover.c, 7
               __libc_start_main()  ../sys-
  deps/generic/libc-start.c, 92

  Stack trace where the error occurred:
                            main()  writover.c, 8
               __libc_start_main()  ../sys-
  deps/generic/libc-start.c, 92

**Memory corrupted.  Program may crash!!**


Exit with return code 0  (0x0000).
  15  total blocks allocated
  0  total blocks freed.
Chaperon searching memory blocks...

End of memory leak processing.
```

READ_UNINIT_MEM

```
#----- readunin.c
int
main()
{
       int x;
       if (3==x) {
              return 5;
       }
       else {
              return 7;
       }
}
#-----
// Chaperon(tm) memory access checker version 2.0  2000-07-07.
// Copyright 1999 BitWagon Software LLC.  All rights reserved.
// Copyright 2000 ParaSoft Corp.          All rights reserved.
[readunin.c:5] **READ_UNINIT_MEM(read)**
>>         if (3==x) {

  Reading uninitialized memory.

  Address:  0xbfffef04
  In block: 0xbfffe000 thru 0xbfffffff (8192 bytes)
                  stack

  Stack trace where the error occurred:
                          main()  readunin.c, 5
               __libc_start_main()  ../sys-
  deps/generic/libc-start.c, 92
```

```
Exit with return code 7  (0x0007).
  14  total blocks allocated
  0  total blocks freed.
Chaperon searching memory blocks...

End of memory leak processing.
Exit 7
```

FREE_DANGLING

```
#----- freedngl.c

#include <stdlib.h>

int
main()
{
       char *p = malloc(13);
       free(p);
       free(p);
       return 0;
}
#-----
// Chaperon(tm) memory access checker version 2.0  2000-07-07.
// Copyright 1999 BitWagon Software LLC.   All rights reserved.
// Copyright 2000 ParaSoft Corp.          All rights reserved.
[freedngl.c:8] **FREE_DANGLING**
>>         free(p);

  Freeing dangling pointer.

  Pointer        : 0x080495c8
  In block       : 0x080495c8 thru 0x080495d4 (13 bytes)
                  block allocated at freedngl.c, 6
                            main()  freedngl.c, 6
               __libc_start_main()  ../sys-
  deps/generic/libc-start.c, 92

stack trace where memory was freed:
                            main()  freedngl.c, 7
               __libc_start_main()  ../sys-
  deps/generic/libc-start.c, 92

  Stack trace where the error occurred:
```

```
                            main()  freedngl.c, 8
               __libc_start_main()  ../sys-
  deps/generic/libc-start.c, 92

**Memory corrupted.  Program may crash!!**


Exit with return code 0  (0x0000).
  15  total blocks allocated
  1  total blocks freed.
Chaperon searching memory blocks...

End of memory leak processing.
```

Summarize leaks

Make sure that your .psrc file has the following line in it:

```
insure++.summarize leaks outstanding
```

Then run Chaperon again.

```
$ $HOME/bin/Chaperon ./writover

// Chaperon(tm) memory access checker version 2.0  2000-07-07.
// Copyright 1999 BitWagon Software LLC.  All rights reserved.
// Copyright 2000 ParaSoft Corp.          All rights reserved.
[writover.c:8] **WRITE_OVERFLOW**
>>         p[11] = 3;

  Writing overflows memory.

          bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
          |              10              | 1 | 1 |
                                             wwwww

   Writing  (w) : 0x0804958b thru 0x0804958b (1 byte)
   To block (b) : 0x08049580 thru 0x08049589 (10 bytes)
                 block allocated at writover.c, 7
                            main()  writover.c, 7
               __libc_start_main()  ../sysdeps/generic/libc-start.c, 92

  Stack trace where the error occurred:
                            main()  writover.c, 8
               __libc_start_main()  ../sysdeps/generic/libc-start.c, 92

**Memory corrupted.  Program may crash!!**
```

```
Exit with return code 0  (0x0000).
  15  total blocks allocated
  0  total blocks freed.
Chaperon searching memory blocks...

End of memory leak processing.
************************** INSURE SUMMARY ********************* v5.2 **
*   Program     : Not available                                       *
*   Arguments   : Not available                                       *
*   Directory   : Not available                                       *
*   Compiled on : Not available                                       *
*   Run on      : July 11, 2000  00:00:00                             *
*   Elapsed time : 00:00:01                                           *
*   Malloc HWM  : 2161 bytes (2K)                                     *
***********************************************************************

MEMORY LEAK SUMMARY
===================

15 outstanding memory references for 2161 bytes (2K).

Leaks detected at exit
----------------------
        10 bytes      1 chunk allocated at writover.c, 7
                            main()  writover.c, 7
               __libc_start_main()  ../sysdeps/generic/libc-start.c, 92


Outstanding allocated memory
----------------------------
       640 bytes      1 chunk allocated at dl-load.c, 538
                  _dl_init_paths()  dl-load.c, 538
                         dl_main()  rtld.c, 632
                _dl_sysdep_start()  ../sysdeps/generic/dl-sysdep.c, 171
                 _dl_start_final()  rtld.c, 232
                       _dl_start()  rtld.c, 193
                      0x40101996()

       556 bytes      1 chunk allocated at dl-minimal.c, 87
                          calloc()  dl-minimal.c, 87
                  _dl_new_object()  dl-object.c, 40
          _dl_map_object_from_fd()  dl-load.c, 795
                  _dl_map_objject()  dl-load.c, 1416
             _dl_map_object_deps()  dl-deps.c, 218
                         dl_main()  rtld.c, 632
                _dl_sysdep_start()  ../sysdeps/generic/dl-sysdep.c, 171
                 _dl_start_final()  rtld.c, 232
                       _dl_start()  rtld.c, 193
                      0x40101996()

       556 bytes      1 chunk allocated at dl-minimal.c, 87
                          calloc()  dl-minimal.c, 87
                  _dl_new_object()  dl-object.c, 40
                         dl_main()  rtld.c, 508
                _dl_sysdep_start()  ../sysdeps/generic/dl-sysdep.c, 171
                 _dl_start_final()  rtld.c, 232
                       _dl_start()  rtld.c, 193
                      0x40101996()
```

```
288 bytes      3 chunks allocated at dl-minimal.c, 87
                   calloc()  dl-minimal.c, 87
   _dl_check_map_versions()  dl-version.c, 273
   _dl_check_all_versions()  dl-version.c, 365
       version_check_doit()  rtld.c, 299
        _dl_receive_error()  dl-error.c, 169
                  dl_main()  rtld.c, 632
         _dl_sysdep_start()  ../sysdeps/generic/dl-sysdep.c, 171
          _dl_start_final()  rtld.c, 232
                _dl_start()  rtld.c, 193
               0x40101996()

 22 bytes      1 chunk allocated at dl-load.c, 312
       add_name_to_object()  dl-load.c, 312
           _dl_map_object()  dl-load.c, 1291
      _dl_map_object_deps()  dl-deps.c, 218
                  dl_main()  rtld.c, 632
         _dl_sysdep_start()  ../sysdeps/generic/dl-sysdep.c, 171
          _dl_start_final()  rtld.c, 232
                _dl_start()  rtld.c, 193
               0x40101996()

 18 bytes      1 chunk allocated at dl-object.c, 41
           _dl_new_object()  dl-object.c, 41
   _dl_map_object_from_fd()  dl-load.c, 795
           _dl_map_object()  dl-load.c, 1416
      _dl_map_object_deps()  dl-deps.c, 218
                  dl_main()  rtld.c, 632
         _dl_sysdep_start()  ../sysdeps/generic/dl-sysdep.c, 171
          _dl_start_final()  rtld.c, 232
                _dl_start()  rtld.c, 193
               0x40101996()

 15 bytes      1 chunk allocated at dl-load.c, 140
           _dl_map_object()  dl-load.c, 140
      _dl_map_object_deps()  dl-deps.c, 218
                  dl_main()  rtld.c, 632
         _dl_sysdep_start()  ../sysdeps/generic/dl-sysdep.c, 171
          _dl_start_final()  rtld.c, 232
                _dl_start()  rtld.c, 193
               0x40101996()

 15 bytes      1 chunk allocated at dl-object.c, 92
           _dl_new_object()  dl-object.c, 92
   _dl_map_object_from_fd()  dl-load.c, 795
           _dl_map_object()  dl-load.c, 1416
      _dl_map_object_deps()  dl-deps.c, 218
                  dl_main()  rtld.c, 632
         _dl_sysdep_start()  ../sysdeps/generic/dl-sysdep.c, 171
          _dl_start_final()  rtld.c, 232
                _dl_start()  rtld.c, 193
               0x40101996()

 12 bytes      1 chunk allocated at dl-deps.c, 437
      _dl_map_object_deps()  dl-deps.c, 437
                  dl_main()  rtld.c, 632
         _dl_sysdep_start()  ../sysdeps/generic/dl-sysdep.c, 171
         _dl_start_final()  rtld.c, 232
               _dl_start()  rtld.c, 193
              0x40101996()

12 bytes      1 chunk allocated at dl-load.c, 528
          _dl_init_paths()  dl-load.c, 528
                 dl_main()  rtld.c, 632
        _dl_sysdep_start()  ../sysdeps/generic/dl-sysdep.c, 171
         _dl_start_final()  rtld.c, 232
               _dl_start()  rtld.c, 193
              0x40101996()

 9 bytes      1 chunk allocated at dl-object.c, 41
          _dl_new_object()  dl-object.c, 41
                 dl_main()  rtld.c, 508
        _dl_sysdep_start()  ../sysdeps/generic/dl-sysdep.c, 171
         _dl_start_final()  rtld.c, 232
               _dl_start()  rtld.c, 193
              0x40101996()

 8 bytes      1 chunk allocated at dl-sysdep.c, 292
    _dl_important_hwcaps()  ../sysdeps/generic/dl-sysdep.c, 292
          _dl_init_paths()  dl-load.c, 524
                 dl_main()  rtld.c, 632
        _dl_sysdep_start()  ../sysdeps/generic/dl-sysdep.c, 171
         _dl_start_final()  rtld.c, 232
               _dl_start()  rtld.c, 193
              0x40101996()
```

Using Chaperon With gdb

Chaperon can be used with your existing gdb, or with a modified gdb-5.0, provided by ParaSoft in `INSTALLDIR/bin.linux2/gdb.exe`.

Either version can be used to set a breakpoint in `_Insure_trap_error`, which allows you to stop program execution at a point where a memory reference error is detected, and examine program state, values of variables, etc.

The ParaSoft version of gdb could also be used to set breakpoints in your binary and execute gdb commands, such as `next`, `step`, `continue`.

Note: Your existing gdb will be able to set breakpoints in your executable as well, but you will not be able to properly continue execution after the breakpoint.

For example:

```
$ gdb.exe Chaperon
```

```
GNU gdb 5.0  extended 2000-09-12 by ParaSoft Corporation for Chaperon on Linux
        x86
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License ...
(gdb) b gdb_setup
Breakpoint 1 at 0x1700
(gdb) run ./chaptest
Starting program: Chaperon ./chaptest
// Chaperon(tm) memory access checker version 2.0  2000-09-25.
// Copyright 1999 BitWagon Software LLC.  All rights reserved.
// Copyright 2000 ParaSoft Corp.          All rights reserved.


Breakpoint 1, 0x1700 in gdb_setup () from ./chaptest
(gdb) b main
Breakpoint 2 at 0x80483d6: file chaptest.c, line 4.
(gdb) disable 1
gdb) cont


Continuing.


Breakpoint 2, main () at chaptest.c:4
4           char *p = (char *)malloc(20);
(gdb) n
6           if (p[1]) {
(gdb) b _Insure_trap_error
Breakpoint 3 at 0x1706
(gdb) c
Continuing.
[chaptest.c:6] (Thread 0) **READ_UNINIT_MEM(read)**
>>     if (p[1]) {


  Reading uninitialized memory.


  Address:  0x080495b1
  In block: 0x080495b0 thru 0x080495c3 (20 bytes)
                  block allocated at chaptest.c, 4
                            main()  chaptest.c, 4


  Stack trace where the error occurred:
                            main()  chaptest.c, 6



Breakpoint 3, 0x1706 in _Insure_trap_error () from ./chaptest


(gdb) where
#0  0x1706 in _Insure_trap_error () from ./chaptest
#1  0x80483e9 in main () at chaptest.c:6
#2  0x4012d9cb in __libc_start_main (main=0x80483d0 <main>, argc=1,
        argv=0xbffff7b4, init=0x8048298 <_init>,
fini=0x804845c <_fini>, rtld_fini=0x4010ae60 <_dl_fini>,
        stack_end=0xbffff7ac) at ../sysdeps/generic/libc-start.c:92
(gdb) quit
```

Finally, you can download the gdb-5.0.patch file, which contains diffs against the original gdb-5.0 source, at at:
`http://www.parasoft.com/products/insure/manuals/v5_2/unix/users/chapsamp.html`

Insure++ Reports

The error reports that have already been shown indicate that Insure++ provides a great deal of information about the problems encountered in your programs. It also provides many ways of customizing the presentation of this information to suit your needs.

Default behavior

By default, Insure++ adopts the following error reporting strategy:

- Error messages are "coded" by a single word shown in uppercase, such as `HEAP_CORRUPT`, `READ_OVERFLOW`, `LEAK_SCOPE`, etc.
- Messages about error conditions are displayed unless they have been suppressed by default or in a site-specific configuration file. (See "Error Codes" on page 171 for a list.)
- Only the first occurrence of a particular (unsuppressed) error at any given source line is shown. (See "Report summaries" on page 72 for ways to change this behavior.)
- Error messages are displayed on output stream `stderr`. (See "The report file" on page 60.) To view error messages on the Insra GUI, see "Sending messages to Insra" on page 83.
- Each error shows a stack trace of the previous routines, displayed all the way back to your `main` program.

The report file

Normally, error reports are displayed on the UNIX `stderr` I/O stream. Users interested in sending output to Insra should consult the Insra section of this manual, which begins on page 79. If you wish to capture both your program's output and the Insure++ reports to a file, you can use the normal shell redirection method. An alternative is to have Insure++ redirect only its output directly by adding an option similar to

```
insure++.report_file bugs.dat
```

to your `.psrc` file. This tells Insure++ to write its reports to the file `bugs.dat`, while allowing your program's output to display as it normally would. Whenever this option is in effect you will see a "report banner" similar to

```
** Insure++ messages will be written to bugs.dat **
```

on your terminal when your program starts to remind you that error messages are being redirected. To suppress the display of this banner add the option

```
insure++.report_banner off
```

to your `.psrc` file.

Normally the report file is overwritten each time your program executes, but you can force messages to be appended to an existing file with the command

```
insure++.report_overwrite false
```

If you wish to keep track of the reports from multiple runs of your code, an alternative is to have Insure++ automatically generate filenames for you based on a template that you provide. This takes the form of a string of characters with tokens such as "`%d`", "`%p`", or "`%V`" embedded in it. Each of these is expanded to indicate a certain property of your program as indicated in the table on page 134.

Thus, for example, the option

```
insure++.report_file %v-errs.%D
```

when executed with a program called `foo` at 10:30 a.m. on the 21st of March 1997, might generate a report file with the name

```
foo-errs.19970321103032
```

(The last two digits are the seconds after 10:30 on which execution began.)

Note that programs which fork will automatically have a “`-%n`” added to their format strings unless a `%n` or `%p` token is explicitly added to the format string by the user. This ensures that output from different processes will always end up in different report files.

You can also include environment variables in these filenames so that

```
$HOME/reports/%v-errs.%D
```

generates the same filename as the previous example, but also ensures that the output is placed in the `reports` sub-directory of the user’s `HOME`.

This method is very useful for keeping track of program runs during development to see how things are progressing as time goes on.

Customizing the output format

By default, Insure++ displays a particular banner for each error report, which contains the filename and line number containing the error, and the error category found, e.g.,

```
[foo.c:10] **READ_UNINIT_MEM(copy)**
```

You can modify this format to suit your aesthetic tastes, or for some other purpose such as enabling the editor in your integrated environment to search for the correct file and line number for each error.

Customization of this output is achieved by setting the `error_format` option in your `.psrc` file to a string of characters containing embedded tokens which represent the various pieces of information that you might wish to see. (A complete list is shown on page 145.)

For example, the command

```
insure++.error_format "\"%f\", line %l: %c"
```

would generate errors in the format

```
"foo.c", line 8: READ_UNINIT_MEM(copy)
```

which is a form recognized by editors such as GNU emacs.

Notice how the embedded double quote characters required backslashes to prevent them from being interpreted as the end of the format string.

A multi-line format can also be generated with a command such as

```
insure++.error_format "%f, line %l\n\t%c"
```

which might generate

```
foo.c, line 8
      READ_UNINIT_MEM(copy)
```

Displaying process information

When using Insure++ with programs that run on remote machines (e.g., in client-server mode) or which fork into multiple processes, you might wish to display additional process-related information in your error reports.

For example, adding the option

```
insure++.error_format \
      "%f, line %l: \n\tprocess %p@%h: %c"
```

to your `.psrc` file would generate errors in the form

```
foo.c, line 8:
      process 1184@gobi: READ_UNINIT_MEM(copy)
```

which contains the name of the machine on which the process is running and its process ID.

Displaying the time at which the error occurred

Especially when using Insure++ with applications that run for a long period, it is often convenient to know exactly when various errors occurred. You can extend the error reports generated by Insure++ in this fashion by adding the ‘`%d`’ and/or

'%t' characters to the error report format as specified in your .psrc file. For example, the format

```
insure++.error_format "%f:%l, %d %t <%c>"
```

generates error reports in the form

```
foo.c:8, 9-Jan-1997 14:24:03 <READ_NULL>
```

Displaying repeated errors

The default configuration suppresses all but the first error of any given kind at a source line. You can display more errors by modifying the parameter report_limit in the .psrc file in either your working or HOME directory.

For example, adding the line

```
insure++.report_limit 5
```

to your .psrc file will show the first five errors of each type at each source line.

Setting the value to zero suppresses any messages except those shown in summaries. (See "Report summaries" on page 72.)

Setting the report_limit value to -1 shows all errors as they occur.

Note that not all information is lost by showing only the first (or first few) errors at any source line. If you enable the report summary (See page 73) you will see the total number of each error at each source line.

Limiting the number of errors

If your program is generating too many errors for convenient analysis, you can arrange for it to exit (with a non-zero exit code) after displaying a certain number of errors by adding the line

```
insure++.exit_on_error number
```

to your `.psrc` file and re-running the program. After the indicated number of errors, the program will exit. If `number` is less than or equal to zero, all errors are displayed.

Changing stack traces

There are two potential modifications you can make to alter the appearance of the stack tracing information presented by Insure++ to indicate the location of an error.

By default, Insure++ will read your program's symbol table at start-up time to get enough information to generate stack traces. To get file and line information, you will need to compile your programs with debugging information turned on (typically via the `-g` switch). If this is a problem, Insure++ can generate its own stack traces for files compiled with Insure++. You can select this mode by adding the options

```
insure++.symbol_table off
insure++.stack_internal on
```

to your `.psrc` file. The `stack_internal` option will take effect after you recompile your program (see page 151), while the `symbol_table` option can be toggled at runtime (see page 164). In this case, the stack trace will display

```
** routines not compiled with insure **
```

in place of the stack trace for routines which were not compiled with Insure++. This will also make your program run faster, particularly at start-up, since the symbol table will not be read.

If your program has routines that are deeply nested, you may see very long stack traces. You can reduce the amount of stack tracing information made available by adding an option such as

```
    insure++.stack_limit 4
```

to your `.psrc` file. If you run your program again, you will see at most[1] the last four levels of the stack trace with each error.

The value “`0`” is valid and effectively disables tracing.

The value “`-1`” is the default and indicates that the full stack trace should be displayed, regardless of length.

Stack traces are also presented to show the function calling sequence when blocks of dynamically allocated memory were allocated and freed. In a manner similar to the `stack_limit` option, the `malloc_trace` and `free_trace` options control how extensive these stack traces are.

Searching for source code

Normally, Insure++ remembers the directory in which each source file was compiled and looks there when trying to display lines of source code in error messages. Occasionally your source code will no longer exist in this directory, possibly because of some sophisticated “build” or “`make`” process.

[1] It is “at most” because some of the lower levels of the trace may be hidden internally by Insure++ and not displayed by default. These levels are still counted for the `report_limit` option.

You can give Insure++ an alternative list of directories to search for source code by adding a line such as

```
insure++.source_path .:$HOME/src:/usr/local/src
```

to the `.psrc` file in your current working or `HOME` directories.

The list may contain any number of directories separated by either spaces or colons (`:`).

Insure++'s error messages normally indicate the line of source code responsible for a problem on the second line of an error report, after the ">>" mark. If this line is missing from the report, it means that the source code could not be found at runtime.

Suppressing error messages

The previous sections discussed issues which can affect the appearance of particular error messages. Another alternative is to completely suppress error messages of a given type which you either cannot, or have no wish to, correct.

The simplest way of achieving this is to add lines similar to

```
insure++.suppress EXPR_NULL, PARM_DANGLING
```

to your `.psrc` file and re-run the program. No suppressed error messages will be displayed, although they will still be counted and displayed in the report summary. (See page 73.)

In this context, certain wild-cards can be applied so that, for instance, you can suppress all memory leak messages with the command

```
insure++.suppress LEAK_*
```

You can suppress all errors with the command

```
insure++.suppress *
```

which has the effect of only creating an error summary. If the error code has sub-categories, you can disable them explicitly by listing the sub-category codes in parentheses after the name, e.g.,

```
insure++.suppress BAD_FORMAT(sign, compatible)
```

Alternatively,

```
insure++.suppress BAD_FORMAT
```

suppresses all sub-categories of the specified error class.

Suppressing error messages by context

The commands described in the previous section either suppress or enable errors in a given category regardless of where in your program the error occurs. This syntax can be extended to specifying particular routines that must appear in the function call stack at the time of the error for it to be enabled or suppressed.

For example, the command

```
insure++.suppress READ_NULL { sub* * }
```

suppresses messages of the given category that occur in any routine whose name begins with the characters “sub”.

The interpretation of this syntax is as follows:

- The stack context is enclosed by a pair of braces.
- Routine names can either appear in full or can contain the ‘*’ or ‘?’ wildcard characters. The former matches any string, while the latter matches any single character.
- An entry consisting of a single ‘*’ character matches any number of functions, with any names.
- Entries in the stack context are read from left to right with the leftmost entries appearing lowest (or most recently) in the call stack.

With these rules in mind, the previous entry is read as

- The lowest function in the stack trace (i.e., the function generating the error message) must have a name that begins with “sub” followed by any number of other characters.
- Any number of functions of any name may appear higher in the function call stack.

A rather drastic, but common, action is to suppress any errors generated from within calls to the X Window System libraries. If we assume that these functions have names that begin with either “X” or “_X”, we can achieve this goal with the statements

```
insure++.suppress all { * X* * }
insure++.suppress all { * _X* * }
```

which suppress errors in any function (or its descendents) that begins with either of the two sequences.

As a final example, consider a case in which we are only interested in errors generated from the routine `foobar` or its descendents. In this case, we can combine `suppress` and `unsuppress` commands as follows

```
insure++.suppress all
insure++.unsuppress all { * foobar * }
```

Suppressing error messages by file/line

In addition to suppressions based on stack traces, you can suppress error messages based on the file/line generating the message.

The syntax for this type of suppression is:

```
file:line#
in file
```

Examples:

```
suppress readbadindex at foo.h:32
```

This suppresses readbadindex error messages at line 32 of foo.h at both compile-time and run-time.

```
suppress parserwarning in header.h
```

This suppresses all parser warnings in header.h.

Wildcards are not supported in filenames for this syntax. However, this syntax can be used at both compile-time and run-time (unlike stack trace suppressions, which can only be used at run-time).

It is illegal to have both a stack trace suppression and a file/line suppression on the same line, e.g.:

```
suppress myerror {a b c} at foo.c:3
```

Suppressing C++ warning messages

The warning messages that Insure++ displays during parsing of C++ code (see page 24) can easily be suppressed if the user does not wish to correct the code immediately. For example, to suppress the warning from that section, simply add

```
insure++.suppress_warning 13-2
```

to your `.psrc` file and recompile. The warning messages will no longer be displayed.

Suppressing other warning messages

For other compile time warning messages that do not have a number associated, there is another suppress option available. The `suppress_output` option takes a string as an argument and will suppress any message that includes text that matches the string. For example, the option

```
insure++.suppress_output wrong arguments passed
```

would suppress the warning from the previous section, as well as any others that included this text string.

Enabling error messages

Normally, you will be most interested in suppressing error messages about which you can or wish to do nothing. Occasionally, however, you will want to enable one of the options that is currently suppressed, either by system default (See "Error Codes" on page 171) or in one of your own `.psrc` files. This is achieved by adding a line similar to the following to your `.psrc` file.

```
insure++.unsuppress RETURN_FAILURE
```

Opaque pointers

You can prohibit Insure++ from checking a pointer by declaring it "opaque". You can do this for a function return value by using the `assert_ok .psrc` option (see page 153) or, more generally, by using an `iic_opaque` function in an interface. When you declare opaque, Insure++ will not check this pointer or any pointers that are derived from it. This is normally done only for a third-party function that returns a pointer to a memory block allocated in a way that will not be seen by Insure++. This option tells Insure++ to ignore such a pointer.

Report summaries

Normally, you will see error messages for individual errors as your program proceeds. Using the other options described so far, you can enable or disable these errors or control the exact number seen at each source line.

This technique is most often used to systematically track down each problem, one by one.

It is often useful, however, to obtain a summary of the problems remaining in a piece of code in order to track its progress.

Insure++ supports the following types of summary reports

- A “bug” summary, which lists all outstanding bugs according to their error codes.
- A “leak” summary, which lists all memory leaks - i.e., places where memory is being permanently lost.
- An “outstanding” summary, which lists all outstanding memory blocks - i.e., places where memory is not being freed, but is not leaked because a valid pointer to the block still exists.
- A “coverage” summary, which indicates how much of the application’s code has been executed.

TCA

None of these is displayed by default.

The “bugs” summary

This report summary is enabled by adding the option

```
insure++.summarize bugs
```

to your `.psrc` file and re-running your program.

In addition to the normal error reports, you will then also see a summary such as the one shown in Figure 7.

```
******************** INSURE SUMMARY ************* v5.2 **
    Program          : gs
    Arguments        : golfer.ps
    Directory        : /usr/home/trf/gsb
    Compiled on      : Jul 15, 1999 15:40:37
    Run on           : Jul 15, 1999 15:44:29
    Elapsed time     : 00:01:06
***********************************************************

PROBLEM SUMMARY - by type
===============

    Problem          Detected   Suppressed
    --------------------------------------------
    EXPR_BAD_RANGE        7          0
    READ_UNINIT_MEM       23         0
    BAD_DECL              1          0
    --------------------------------------------
    TOTAL                 31         0
    --------------------------------------------

PROBLEM SUMMARY - by location
===============

EXPR_BAD_RANGE: Expression exceeded range, 7 occurrences
    5 at ialloc.c, 170
    1 at ialloc.c, 176
    1 at ialloc.c, 182

READ_UNINIT_MEM: Reading uninitialized memory, 23 occurrences
    7 at gxcpath.c, 137
    7 at gxcpath.c, 241
    1 at gdevx.c, 424
    1 at gdevxini.c, 213
    2 at gdevxini.c, 221
    1 at gdevxini.c, 358
    1 at gdevxini.c, 359
    1 at gdevxini.c, 422
    1 at gdevxini.c, 454
    1 at gdevxini.c, 514

BAD_DECL: Global declarations are inconsistent, 1 occurrence
     1 at gdevx.c, 93
```

Figure 7. Sample “bugs” report summary

The first section is a header that indicates the following information about the program being executed:

- The name of the program
- Its command line arguments, if available
- The directory from which the program was run
- The time it was compiled
- The time it was executed
- The length of time to execute

This information is provided so that test runs can be compared accurately as to the arguments and directory of test. The time and date information is supplied to correlate with bug tracking software.

The second section gives a summary of problems detected according to the error code and frequency. The first numeric column indicates the number of errors detected but not suppressed. This is the total number of errors, which may differ from the number reported, since, by default, only the first error of any particular type is reported at each source line. The second column indicates the number of bugs that were not displayed at all due to “suppress” commands.

The third section gives details of the information presented in the second section, broken down into source files and line numbers.

The “leak” summaries

The simplest memory leak summary is enabled by adding the line

```
insure++.summarize leaks outstanding
```

to your `.psrc` file and re-running your program.

The output indicates the memory (mis)use of the program, as shown in Figure 8.

```
********************* INSURE SUMMARY ************** v5.2 **
    Program          : leak
    Arguments        :
    Directory        : /usr/home/whicken/test
    Compiled on      : Jul 15, 1999 15:09:05
    Run on           : Jul 15, 1999 15:09:31
    Elapsed time     : 00:00:02
*************************************************************
MEMORY LEAK SUMMARY
===================

4 outstanding memory references for 45 bytes.

Leaks detected during execution
-------------------------------
    10 bytes   1 chunk    allocated at leak.c, 6

Leaks detected at exit
----------------------
    10 bytes   2 chunk    allocated at leak.c, 7

Outstanding allocated memory
----------------------------
    15 bytes   1 chunk    allocated at leak.c, 8
```

Figure 8. Sample "leaks" report summary

The first section summarizes the "memory leaks" that were detected during program execution, while the second lists leaked blocks that were detected at program exit. These are potentially serious errors, in that they typically represent continuously increasing use of system resources. If the program is "leaking" memory, it is likely to exhaust the system resources eventually and will probably crash. The first number displayed is the total amount of memory lost at the indicated source line, and the second is the number of chunks of memory lost. Note that multiple chunks *of different sizes* may be lost at the same source line - depending on which options you are using. To customize the report, there are three `.psrc` options available: `leak_combine`, `leak_sort`, and `leak_trace`.

The `leak_combine` option controls how Insure++ merges information about multiple blocks. The default behavior is to combine all information about leaks that were allocated from locations with identical stack traces (`leak_combine` trace). It may be that you would rather combine all leaks based only on the file and line they were allocated, independent of the stack trace leading to that allocation. In that case, you would use leak_combine location. Or, you may simply want one entry for each leak (leak_combine none).

The leak_sort option controls how the leaks are sorted, after having been combined. The options are none, location, trace, size, and frequency (size is the default). Sorting by size lets you look at the biggest sources of leaks, sorting by frequency lets you look at the most often occurring source of leaks, and sorting by location provides an easy way to examine *all* your leaks.

The leak_trace option causes a full stack trace of each allocation to be printed, in addition to the actual file and line where the allocation occurred (this replaces the detailed modifier from earlier versions of Insure++). This option will not work if the malloc_trace option is non-zero (see page 160).

The third section shows the blocks that are allocated to the program at its termination and which have valid pointers to them. Since the pointers still allow the blocks to be freed by the program (even though they are not), these blocks are not actually leaked. This section is only displayed if the outstanding keyword is used. Normally, these blocks do not cause problems, since the operating system will reclaim them when the program terminates. However, if your program is intended to run for extended periods, these blocks are potentially more serious.

The “coverage” summary

TCA

The coverage summary is enabled by adding the line

```
    insure++.summarize coverage
```

to your .psrc file and re-running your program.

In addition to the normal error reports, you will see a summary indicating how much of the application’s source code has been tested. The exact form of the output is controlled by the .psrc file option coverage_switches, which specifies the command line switches passed to the `tca` command to create the output.

If this variable is not set, it defaults to

```
insure++.coverage_switches -dS
```

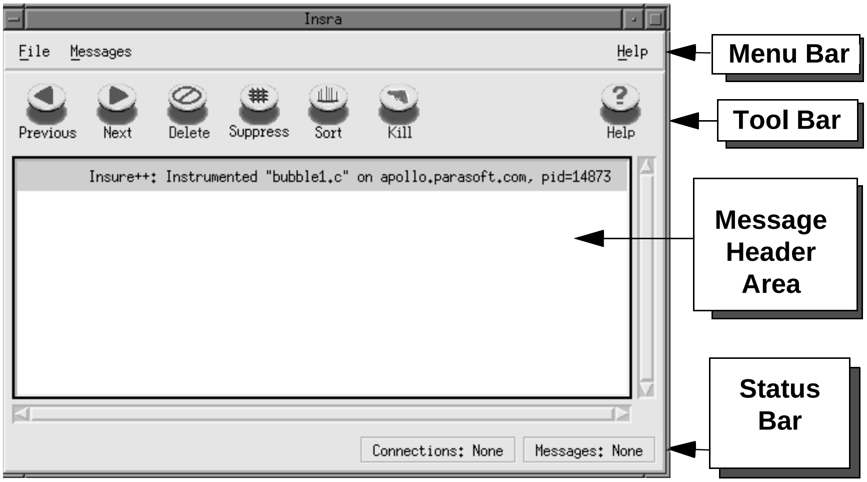
which displays an application level summary of the test coverage such as

```
COVERAGE SUMMARY
================
      11   blocks untested
      42   blocks tested

78% covered
```

For details on the formatting of this output using the `coverage_switches` option, consult the manual page for the `tca` command or the on-line man pages with the command “`man tca`”.

Insra

Insra is a graphical user interface that displays error messages generated by Insure++*.* The messages are summarized in a convenient display, which allows the developer to navigate quickly through the list of bug reports and violation messages, suppress messages, invoke an editor for immediate corrections to the source code, and delete messages as bugs are fixed.



Insure++: Instrumented "bubble1.c" on apollo.parasoft.com, pid=14873

Connections: None

Messages: None

Figure 9. Initial Insra display

The Insra display

Status Bar

During compilation/run time, Insure++ makes a connection to Insra each time an error is detected. The status bar will report the number of error messages currently being displayed and the number of active connections. An active connection is denoted by a yellow star to the left of the session header. A connection will remain active as long as the program is still compiling/running. Insra will not allow you to delete a session header as long as its connection remains active, and you may not exit Insra until all connections have been closed.

Tool Bar

The tool bar allows the user to:

- Scroll up and down through messages
- Delete selected messages as bugs are fixed
- Suppress errors detected by Insure++
- Sort messages by order (time) reported, error category, or directory and file
- Kill the selected active connection

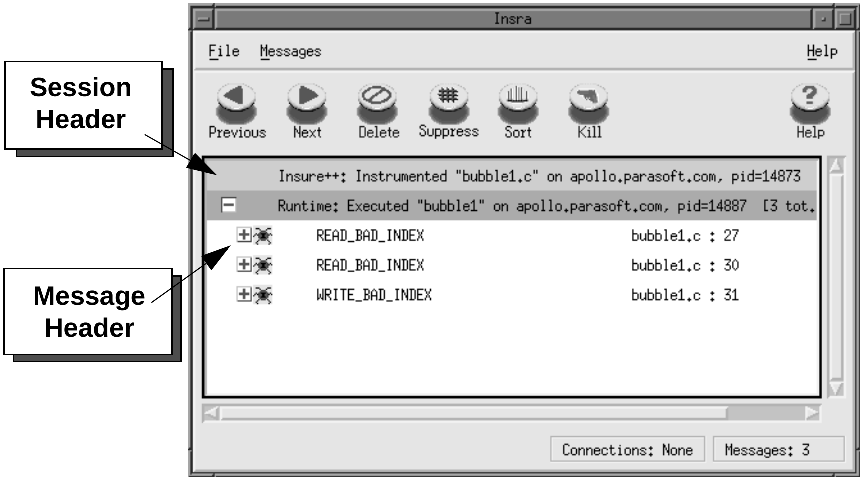
Message Header Area

The message header area contains session headers and message headers for programs currently connected to Insra. To see an example of Insra displaying Insure++ error messages, please refer to Figure 10 on the following page.

Session Header

When the first error is detected for a particular compilation or execution, a session header is sent to Insra. The session header includes the following information:

- Compilation/execution
- Source file/program
- Host on which the process is running
- Process ID



```
Insure++: Instrumented "bubble1.c" on apollo.parasoft.com, pid=14873
Runtime: Executed "bubble1" on apollo.parasoft.com, pid=14887  [3 tot.
    READ_BAD_INDEX                       bubble1.c : 27
    READ_BAD_INDEX                       bubble1.c : 30
    WRITE_BAD_INDEX                      bubble1.c : 31
```

Connections: None   Messages: 3

Figure 10. Sample Insra display with messages

Message Header

There are several types of message headers.

Messages generated by Insure++ include:

- Error Category, e.g. LEAK_SCOPE
- File name
- Line number

Message headers will also appear for various summary reports generated by Insure++. These reports are generated using `.psrc` options. (See “Report Summaries” on page 72.) Double-clicking a message header will open up the message window for the error or summary report selected.

Message Window

The message window opens when you double-click a message header. This window contains the error message or summary report for the selected message header.

Error Message

Insure++ error messages include:

- Line of source code where the error occurred;
- Explanation of the error detected;
- Stack traces for quick reference to the original source.

The stack traces are “live” and can be double-clicked to launch an editor for viewing and correcting the indicated line of code. See “Viewing source files” on page 88.

All messages sent to Insra are marked with a special icon. Please refer to the following table for a brief description of each icon.

Icon	Explanation
🐞	Insure++ error message
📄	Insure++ summary report
💧	Memory leak
⚡	Trapped signal

Sending messages to Insra

By default, all Insure++ output is sent to `stderr`. To redirect messages to Insra, simply add the following line to your .psrc file.

```
insure++.report_file insra
```

This will redirect both compile-time and run-time messages to Insra.

The option

```
    insure++.runtime.report_file insra
```

will send only runtime messages to Insra. Compile-time messages will continue to be sent to `stderr`.

The option

```
    insure++.compile.report_file insra
```

will send only compile-time messages to Insra. Run-time messages will continue to be sent to `stderr`.

With `insure++.report_file insra` in your `.psrc` file, each time an error is detected, Insure++ attempts to establish a connection to Insra. If Insra is not yet running, it will automatically start. Once the connection is established, a session header and all corresponding message headers will be reported in the order they were detected. Each new compilation or program, with its own session header and messages, will be displayed in the order in which it connected to Insra.

Viewing and navigating

Message headers sent to Insra are denoted by a specific icon (See “The Insra display” on page 80). The body of the currently selected message is displayed in a separate message window. Double-click on the message header to view the message itself. The message header area and the message window are both resizable, and scroll bars are also available to access text that is not visible.

Currently active messages become inactive when they are deleted or suppressed.

Deleting messages

Once error messages have been read and analyzed, the user may wish to clear them from the window. The **Delete** button on the Insra toolbar allows you to remove error messages from the display as errors are corrected in your code. A message or an entire session may be removed from the display by selecting an entry in the message header area and clicking the **Delete** button. A message can also be deleted by selecting **Messages/Delete** from the menu bar.

Suppressing messages

You can easily suppress (turn “off”) error messages that you do not want Insure++ to generate. The suppressions window allows you to insert, modify, and delete suppression options for your Insure++ error messages. The suppression options you choose can be saved into your .psrc files so that they will be used again the next time you use Insure++.

The Toolbar

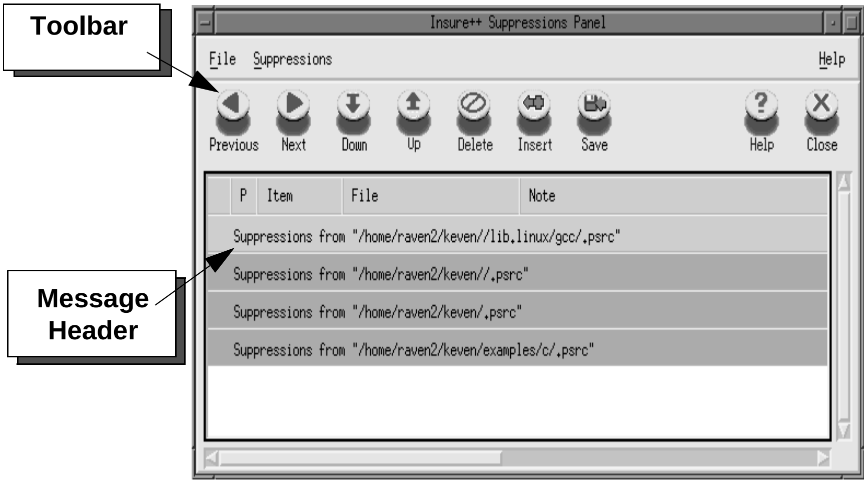
Moving from left to right across the toolbar:

The **Previous** and **Next** arrows select the next higher or next lower suppression option, respectively.

The **Up** and **Down** arrows move the currently selected suppression option up or down in the order of options. Because Insure++ follows suppression options from top to bottom, the order in which they are listed affects the outcome of the suppression. (For more information, see the “Reports” chapter of this user guide.)

The **Delete** button deletes the currently selected suppression option.

The **Insert** button inserts a new suppression option below the currently selected option. If you had an error message selected when you clicked the **Suppress** button on the Insra GUI, a suppression option for that particular error message will be inserted. Otherwise, the default suppression option (suppress all error



```
Suppressions from "/home/raven2/keven//lib.linux/gcc/.psrc"
Suppressions from "/home/raven2/keven//.psrc"
Suppressions from "/home/raven2/keven/.psrc"
Suppressions from "/home/raven2/keven/examples/c/.psrc"
```

Figure 11. Suppressions window

messages) will be inserted. Suppression options are easy to edit. To change an option, simply follow the directions given below.

The **Save** button writes all the suppression options that have been marked as persistent (see below) into the indicated .psrc files (see below). These suppression options will be in effect the next time you use Insure++.

The **Help** button provides context-sensitive help, which in this window means that clicking anywhere will bring up this file.

The **Close** button closes the suppression window.

Editing suppression options

An individual suppression option consists of five parts, listed below from left to right:

1) Suppress/unsuppress

This field specifies whether the error message listed is to be suppressed (as indicated by a speaker with an X through it) or unsuppressed (indicated by a speaker with no X through it.) Double-clicking on the field toggles between suppressing and unsuppressing the error message.

2) Persistence

This field specifies whether the suppression option will be saved to the .psrc file under which it is listed. Double-clicking this field toggles it from persistent (the field is checked) to temporary (the field is unchecked). Options marked as persistent will be added to the appropriate .psrc files when the save button is clicked. Options marked as temporary will be discarded. Options with an X in this field cannot be made persistent, either because they are hardwired or because the files in which they are placed are not writable. New options are marked as persistent by default.

3) Item

Double-clicking this field allows you to type in the name of the error message you would like to suppress or unsuppress. You can use a wildcard ("*") to match all error messages and also suppress and unsuppress by error category and context. For more information on suppressing error messages, see the "Suppressing error messages" section in the "Reports" chapter of this user guide.

4) File

Double-clicking this field allows you to type in the file for which you would like to suppress or unsuppress messages. Entering a blank field will insert a "*" which will match all files.

5) Note

You may use this field to enter your own notes regarding the suppression option listed.

Configuration (.psrc) files

The headers in the window show the various locations in which .psrc files reside. Insra will display the suppression options as read from each file under the appropriate header. When you add a new option using the **Insert** button, it will be inserted below the currently selected option. You can then move it into the file in which you would like it saved, or mark it as temporary by double-clicking the persistence field (see above). There are two special locations where suppression options may reside other than actual .psrc files: hardwired options and command line options. The former are set internally by Insure++, and therefore cannot be permanently changed. They can be edited and/or removed in the window temporarily, however. The latter are options passed using the -Zop and -Zoi options on the Insure++ command line. These options, like hardwired options, cannot be made persistent, but can be moved into a .psrc file if you decide that you want to make them permanent.

Kill process

When an active connection is selected, clicking the **Kill** button will stop the selected compilation or execution.

Viewing source files

You can view the corresponding source file and line number for a particular error message by double-clicking on any line of the stack trace displayed in the message window. In most cases, the file and line number associated with a given message have been transmitted to Insra. If Insra is unable to locate the source file, a dialog box will appear requesting that you indicate the correct source file.

Selecting an editor

In addition to the location of the source file, Insra must also know the name of your editor and the command line syntax in order to display the correct file and line from the original source code.

Insra obtains this information by reading the .psrc option

```
    insra.visual [editor_command]
```

This command may contain the special tokens `%f` and `%l`, which represent the file name and line number, respectively.

The command will then be executed to load the file into your editor. It is most important to include the full path of any binary that lives in a location not pointed to by your PATH environment variable. If the variable has not been set, vi will be used by default.

Some editors are not X applications and must be run in a terminal window. vi requires the following command in order to load the file successfully:

```
    insra.visual xterm -e vi +%l %f
```

Other editors, e.g. Emacs, do not require an external terminal program like xterm when configured for use as an X application. In this case, the command string should be similar to the following:

```
    insra.visual emacs +%l %f
```

Note: Most implementations of vi and Emacs appear to be sensitive to the order of the line number and file name command line arguments, requiring the line number to precede the file name.

Saving/loading messages to a file

All current messages can be saved to a file by selecting **File/Save** or **File/Save As** from the menu bar. A dialog box allows you to select the destination directory and name of the report file. Report files have the default extension `rpt`. After a report file name has been selected, subsequent **File/Save** selections save all current messages into the report file without prompting for a new filename. A previously saved report file can be loaded by selecting **File/Load** from the menu bar. A dialog box then allows you to select which report file to load.

Help

On-line help can be obtained by choosing **Help** from the menu bar. This will provide a list of topics on the use of Insra. In addition to providing help on the various functional pieces of Insra, the `FAQ` text is available for browsing.

Context-sensitive help is accessible by simply clicking the Help button on the toolbar. When selected, the mouse cursor changes to the question mark arrow combination; clicking any visual element of Insra will bring up a help window with a description of that item.

Setting Preferences

You can modify Insra's appearance with .psrc configuration options.

These options are:

`insra.body_background_color [color]`

Specifies the color used for the message body area background. The default is white.

`insra.body_font [font]`

Specifies the font used for the message body text. The default is fixed.

`insra.body_height [number of rows]`

Specifies the starting height of the message window in number of rows of visible text. The default is 8.

`insra.body_text_color [color]`

Specifies the color used for the message body text. The default is black.

`insra.body_width [columns of text]`

Specifies the starting width of the message window in number of columns of visible text. The default is 80, but if this value is set to a different value than header_width then the larger value will be used.

`insra.button_style [round or square]`

Specifies the shape of buttons that will be shown on toolbars. The default is round.

`insra.coloured_shadows [on or off]`

Specifies if round button shadows will be colored the same color as the application background. The default is on.

`insra.expose_on_message [on or off]`

Specifies if the Insra GUI will be placed on top of windows stack if it receives a new message. The default is off.

Note: this option works only in by-time view mode.

`insra.follow_messages [on or off]`

Specifies if the main window messages area will be automatically scrolled to follow arriving messages. The default is off.

Note: this option works only in by-time view mode.

`insra.header_background_color [color]`

Specifies the color used for the message header area background. The default is white.

`insra.header_font [font]`

Specifies the font used for the message header text. The default is fixed.

`insra.header_height [number of rows]`

Specifies the starting height of the message header area in number of rows of visible text. The default is 8.

`insra.header_highlight_color [color]`

Specifies the color used to indicate the currently selected message or session header in the message header area. The default is LightSteelBlue2.

`insra.header_highlight_text_color [color]`

Specifies the color used for the text of the currently selected message or session header in the message header area. The default is black.

`insra.header_session_color [color]`

Specifies the color used to indicate a session header. The default is LightSkyBlue3.

`insra.header_session_text_color [color]`

Specifies the color used for session header text. The default is black.

`insra.header_text_color [color]`

Specifies the color used for message header text. The default is black.

`insra.header_width [number of columns]`

Specifies the starting width of the header area in number of columns of visible text. The default is 80, but if this value is set to a different value than body_width, the larger value will be used.

`insra.mark_unique [on or off]`

Specifies if messages not duplicated across all connections from the same tool have to be marked by a special arrow-like icon. The default is on.

`insra.port [port_number]`

Specifies which port Insra should use to communicate with Insure++ and Insure++-compiled programs. The default is 3255.

`insra.sourcepath [dir_path1, dir_path22,. . . ]`

Specifies directories to be searched by Insra to find source files launching editor or showing source lines

`insra.toolbar [on or off]`

Specifies whether Insra’s toolbar is displayed. All toolbar commands can be chosen from the menu bar. The default is on.

`insra.viewmode [by_error/by_file/by_time/off/tool]`

Specifies initial view mode. Allows to override view mode setting made by tools connecting with Insra.

by_error- Insra GUI is initially set in view-by error-category mode

by_file- Insra GUI is initially set in view-by-file mode

by_time- Insra GUI is initially set in view-by-time mode

off- Insra GUI is initially launched with the default view mode (i.e. by-time mode)

tool- means that view mode will be set by the first tool that connects with the Insra GUI. This is the default setting.

`insra.visual [editor command]`

Specifies how Insra should call an editor to display the line of source code causing the error. Insra will match the `%l` token to the line number and the `%f` token to the file name before executing the command. Setting this option with no command string disables source browsing from Insra. The default is

```
xterm -e vi +%l %f.
```

Troubleshooting

Insra does not start automatically

Symptom:

While compiling or running, your program seems to hang when error output is directed to Insra and Insra is not yet running.

Solution:

Run Insra by hand. Type

```
insra &
```

at the prompt, wait for the Insra window to appear and then run or compile your program again. Output should now be sent to Insra.

Multiple users of Insra on one machine

Symptom:

When more than one user is attempting to send message reports to Insra, messages are lost.

Solution:

Each invocation of Insra requires a unique port number. By default, Insra uses port 3255. If collisions are experienced, e.g. multiple users are on one machine, set the `.psrc` option `insra.port` to a different port above 1024. Ports less than 1024 are reserved for suid-root programs and should not be used with Insra.

Source browsing is not working

Symptom:

```
***Error while attempting to spawn browser
execvp failed!
```

Solution:

Insra attempted to launch your editor to view the selected source file, but could not locate either xterm or your editor on your path. Please make sure that both of these applications are in directories that are on your path or that you call them with their complete pathnames.

Code Insertions

Most programmers write code that makes assumptions about various things that can happen. These assumptions can vary from the very simple, such as “I’m never going to pass a `NULL` pointer to this routine”, to the more subtle, such as “`a` and `b` are going to be positive”.

Whether this is done consciously or not, the problems that occur when these assumptions are violated are often the most difficult to track down. In many cases, the program will run to completion with no indication of error, except that the final answer is incorrect.

Debugging the hard way

The simple case just described, the `NULL` pointer, will probably be tracked down pretty easily, since Insure++ will pinpoint the error immediately. A few minutes of work should eliminate this problem.

The second case is much harder.

One option is to add large chunks of debugging code to your application to check for the various cases that you don’t expect to show up. Of course, you normally have an idea of where the problem is, so you start by putting checks there. You then run the code and sort through the mass of output, trying to see where things started to go awry. If you guessed wrong, you insert more checks in other places of the code and repeat the entire process.

If you are lucky, the code you insert to catch the problem won’t add bugs of its own.

Once you’ve found the problem, you can either remove the debugging code (introducing the possibility of deleting the wrong things and bringing in new bugs) or comment it out for use next time (cluttering the source code).

An easier solution

A second option is to have Insure++ add the checking code to your application automatically and invisibly.

The basic idea is that you tell Insure++ what you’d like to check by providing an “Insure++ interface module”. This can be kept separate from your main application and added and removed at compile time. Furthermore, Insure++ automatically inserts it in every place that you use a particular piece of code so you only have to go through this process once. Finally, errors that are detected are diagnosed in the same way as all other Insure++ errors. You get a complete report of the source file, line number, and function call stack together with any other information that you think is useful.

An example

Assume that you have a routine in your program called `cruncher` which takes three double precision arguments and returns one. For some reason, possibly connected with the details of your application, you expect the following rules to be true when calls are made to this routine:

- The sum of the three parameters is less than 10.
- The first parameter is always greater than zero.
- The return value is never zero.

To enforce these rules with Insure++, you create a file containing the following “code”.

```
/*
 * crun_iic.c
 */
(double a, double b, double c)
{
     double ret;

     if(a+b+c >= 10.) {
          iic_error(USER_ERROR,
10:                 "Sum exceeds 10: %f+%f+%f\n",
11:                 a, b, c);
12:         }
13:         if(a <= 0) {
14:             iic_error(USER_ERROR,
15:                 "a is negative: %f\n", a);
16:         }
17:         ret = cruncher(a, b, c);
18:         if(ret == 0) {
19:             iic_error(USER_ERROR,
20:                 "Return zero: %f,%f,%f => %f\n",
21:                 a, b, c, ret);
22:         }
23:         return ret;
24:     }
```

Note that this looks just like normal C code with the rather strange exception that the routine `cruncher` calls itself at line 17!

This is not normal C code - it’s an Insure++ interface description, and it behaves rather like a complicated macro insertion. Wherever your original source code contains calls to the function `cruncher`, they will be replaced by this set of error checks, and the indicated call to the routine `cruncher`.

The net effect will be as though you had added all this complex error checking and printing code manually, except that Insure++ does it automatically for you. Another advantage is the use of the `iic_error` routine rather than a conventional call to `printf` or `fprintf`. The `iic_error` routine performs the same task--printing data and strings--but also includes in its output information the source file and line number at which the call is being made and a full stack trace.

Using the interface

Once you've written this interface description, using it is trivial. First, you compile it with the special Insure++ interface compiler, iic. If you put the code in a file called crun_iic.c, for example, you would type the command

```
iic crun_iic.c
```

This creates a file called crun_iic.tqs, which is an "Insure++ interface module".

You can use this module in one of two ways.

If you plan to use this interface check on a regular basis during the development of your project, you should insert the line

```
insure++.interface_library crun_iic.tqs
```

in the .psrc file in either your current working or $HOME directory. All future invocations of Insure++ will then insert this interface check.

If you wish to use the interface check intermittently on some of your compiles, you can add the name of the interface module to the Insure command line when you compile and link your source code. For example the command

```
insure -c myfile1.c
```

would become

```
insure crun_iic.tqs -c myfile1.c
```

Note that you can specify more than one interface in any interface file or include multiple interface modules on the interface_library line in your .psrc file or on the insure command line.

Conclusions

This section has shown how you can add your own error checking either to extend or replace the error checking performed automatically by Insure++ by defining interface modules. These are actually a very powerful way of extending the capabilities of Insure++, and are described more fully in “Interfaces” on page 103.

Interfaces

The section “Code Insertions” on page 97, described a way of using Insure++ interface descriptions to add user-level checking to function calls. This usage is only one of the things that interfaces can do to extend the capabilities of Insure++. This section describes the purpose of these interfaces in more detail and also shows you how to write your own.

What are interfaces for?

Interface descriptions provide an extremely powerful facility which allows you to perform extensive checking on functions in system or third party libraries *before* problems cause them to crash.

Most problems encountered in libraries are due to their being called incorrectly from the user application. Insure++ interfaces are designed to trap and diagnose errors where your code makes calls to these functions. This provides the most useful information for correcting the error.

Essentially, an interface is a means of enforcing rules on the way that a function can be called and the side effects it has on memory. Typically, interfaces check that all parameters are of the correct type, that pointers point to memory blocks of the appropriate size, and that parameter values are in correct ranges. Whenever a function is expected to create or delete a block of dynamic memory, interfaces also make calls that allow Insure++’s runtime library to update its internal records.

Writing interfaces for your libraries is a fairly simple task once the basic principles are understood. To help in relating the purpose of an interface to its implementation, the following sections describe two simple examples, one in C and one in C++.

A C example

Consider the following code, which makes a call to a hypothetical library function `mymalloc`. See the file `mymal.c` below for a definition of `mymalloc`.

```
/*
 * File: mymaluse.c
 */
main()
{
    char *p, *mymalloc();

    p = mymalloc(10);
    *p = 0;
    return (0);
}
```

In order to get the best from Insure++, you need to summarize the expected behavior of the `mymalloc` function. For this example, let us assume that we want to enforce the following rules:

- The single argument is an integer that must be positive.
- The return value is a pointer to a block of memory that is allocated by the routine.
- The size of the allocated block is equal to the supplied argument.

To do so, we create a file with the following interface

```
/*
 * File: mymal_i.c
 */
char *mymalloc(int n)
{
    char *retp;
    if(n <= 0)
        iic_error(USER_ERROR,
            "Negative argument: %d\n",n);
    retp = mymalloc(n);
11:        if(retp) iic_alloc(retp, n);
12:        return retp;
13:   }
```

The key features of this code are as follows

Line 4 A standard ANSI function declaration for the function to be described, including its return type and arguments. (Old-style function declarations can also be used.)

Line 7 A check that the argument supplied is positive, as required by the rules that we are trying to enforce. If the condition fails, we use the special `iic_error` function to print an Insure++-style error message, using standard `printf` notation.

Line 10 This (apparently recursive) call to the `mymalloc` function is where the actual call to the function will be made when this interface is expanded. It appears just as in the function declaration.

Line 11 If the return value from the function call is not zero, we use the `iic_alloc` function to indicate that a block of uninitialized memory of the given size has been allocated and is pointed to by the pointer `retp`.

Line 12 The interface description ends by returning the same value returned from the call to the actual function in Line 10.

If you compile and link this interface description into your program (using the techniques described in “Using interfaces” on page 113), Insure++ will automatically check for all the requirements whenever you call the function.

```
/*
 * File: mymal.c
 */
#include <stdlib.h>

char *mymalloc(int n)
{
    return (char *)malloc(n);
}
```

A C++ example

```
/*
 * File: bag.h
 */
class Bag {
    struct store {
        void *ptr;
        store *next;
    };
    store *top;
    public:
    Bag() : top(0) { }
    void insert(void *ptr);
};
```

```
/*
 * File: bag.C
 */
#include “bag.h”

int main(void) {
    Bag bag;

    for (int i = 0; i < 10; i++) {
        int *f = new int;
        bag.insert(f);
    }
    return 0;
}
```

```
/*
 * File: bagi.C
 */
#include “bag.h”

```

```
void Bag::insert(void *ptr) {
     store *s = new store;
     s->next = top;
     top = s;
     s->ptr = ptr;
     return;
}
```

Let’s assume that `bagi.C` is a part of a class library that was not compiled with Insure++, e.g. a third-party library. We can simulate this situation by compiling the files with the following commands:

```
insure -g -c bag.C
CC -c bagi.C
insure -g -o bag bag.o bagi.o
```

An interface for the `insert` class function might look like this:

```
/*
 * File: bag_i.C
 */
#include “bag.h”

void Bag::insert(void *ptr) {
     iic_save(ptr);
     insert(ptr);
     return;
}
```

We can then compile the interface file with the `iic` compiler as follows:

```
iic bag_i.C
```

To get Insure++ to use the new interface description, we need to use the following compilation commands in place of the earlier commands:

```
CC -c bagi.C
insure -g -o bag bag.C bagi.o bag_i.tqs
```

The basic principles of interfaces

As shown in the previous examples, interface descriptions have the following elements:

- The declaration of the interface description looks just like a piece of C code for the described function. It declares the arguments and return type of the function. Either ANSI or Kernighan & Ritchie style declarations may be used, but ANSI style is preferred, since K&R style declarations have implicit type promotions.
- The body of the interface description uses calls to functions whose names start with `iic_` to describe the behavior of the routine.
- The interface function appears to call itself at some point.

These concepts are common to all interface descriptions.

Interface creation strategy

There are several possible strategies for creating interfaces for your software depending on what resources you have available and how much time you wish to spend on the project.

Normally, we recommend the following steps:

- Create a file containing ANSI-style prototypes for the functions for which you want to make interfaces.
- Extend these prototypes by adding additional error checks with the built-in `iic_` functions.

Getting to the first stage will allow you to perform strong type-checking on all the functions in your application. Going to the second stage provides full support for all of Insure++’s error checking capabilities.

Various aids are provided to help you implement these two stages, as briefly summarized in the flowchart in Figure 12, which includes page references for the most important steps.

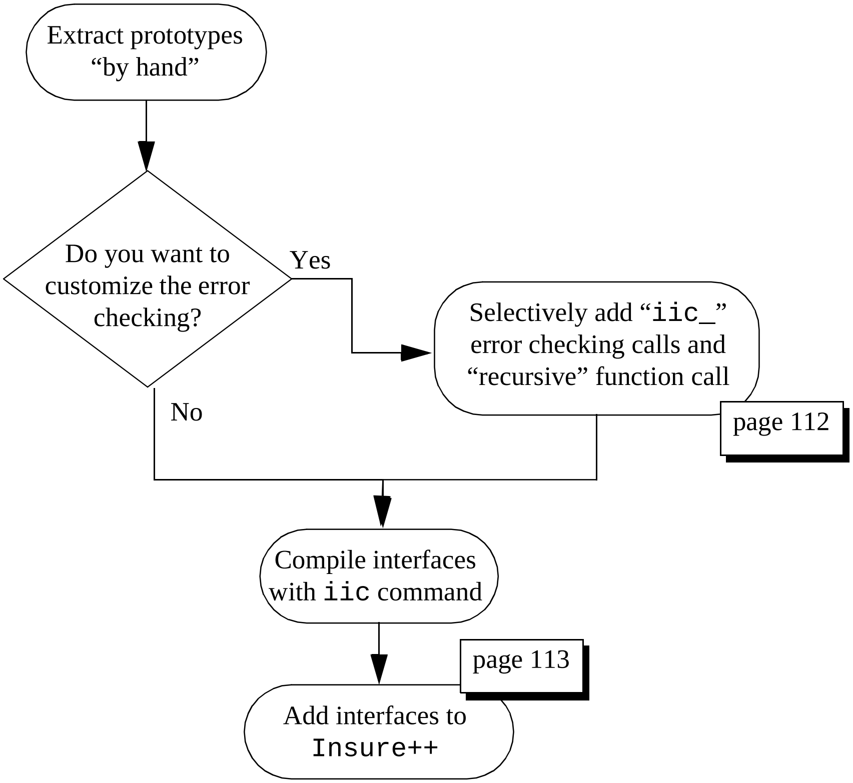

Extract prototypes "by hand"
Do you want to customize the error checking?
Yes
No
Selectively add "iic_" error checking calls and "recursive" function call
page 112
Compile interfaces with iic command
page 113
Add interfaces to Insure++

Figure 12. Strategy for creating interfaces

Trivial interfaces - function prototypes

The interfaces described so far have been “complete” in the sense that they contain error checking calls and also the “fake” recursive call typical of an interface function. There is actually one level of interface that is even simpler than this--an ANSI-style function prototype.

If you make a file containing ANSI-style prototypes for all of your functions, compile it with the `iic` program, and then add it to your `insure` command (as described on page 113), you will get strong type-checking for all of your functions. You can then incrementally add to this file the extended interface descriptions with better memory checking and the “fake” recursive call.

Using `iiwhich` to find an interface

The simplest way to generate an interface is to copy one from a routine that does something similar. In the two examples which started this section, we used interfaces to functions that behaved roughly the same way that `malloc` and `memcpy` operate. Furthermore, these two system functions are ones that Insure++ knows about automatically, because interfaces to all system calls are shipped with Insure++.

To see how their interfaces are defined, we use the command `iiwhich` as follows

```
iiwhich malloc memcpy
```

The output from this command is shown in Figure 13.

Note that on your system, these may be linkable interfaces. If this is the case, you will need to find the source code to these interfaces in the `src.$ARCH/$COMPILER` directory).

```
malloc: Interface in
/usr/local/release/lib.solaris/cc/libc.tqi

     void *malloc(size_t size)
     {
          void *r;

          ;
          if (size < 0)
               iic_error(USER_ERROR, “Negative size
               passed to malloc: %ld”,
                    (long) size);
          r = malloc(size);
          if (!r) {
               if (!_Insight_is_direct_linked()) {
                    iic_error(RETURN_FAILURE,
                    “malloc(%u) failed: %s”,
                         size, sys_errlist[errno]);
               }
          } else {
               ;
               iic_alloc(r, size);
               iic_write_uninit_pattern(r, size);
          }
          ;
          return r;
     }

memcpy: *Linkable* interface in
/usr/local/release/lib.solaris/cc/libc.tqi

     void * memcpy(void *d, void *s, size_t len);
```

Figure 13. Insure++ interfaces for `malloc` **and** `memcpy`

The first block of “code” is the interface which defines the behavior of the malloc function, and the second describes memcpy. Note that they both follow the principles described above. They look more or less like C code with one strange exception: each function appears to call itself!

This is not recursive behavior, because this is not real C code. What really happens is that calls to the functions shown are *replaced* by the interface code. Nonetheless, it can be thought of as C code when you write your own interfaces.

A second slightly tricky feature concerns the behavior of function calls made within an interface definition. These are of two types:

- Calls to Insure++ interface functions, whose names begin with iic_, are detected by the iic command and turned into sequences of error checking calls. They are not real function calls in themselves.
- Calls to other functions are made exactly as requested, *with no additional error checking*. This can be a problem if you end up passing a bad pointer to an unchecked library call, which may cause the program to fail before Insure++ can print an error message.

Note that the iiwhich command is also useful if you want to see what properties of a function are being checked by Insure++, or if Insure++ knows anything about it. The command

```
iiwhich foo
```

shows you the interface for the function foo, if it exists. If no interface exists, no checking will be done on calls to this function unless you write an interface yourself.

Writing simple interfaces

Using iiwhich can save you a lot of time. Before starting to write your own interface files, particularly for system functions, you should check that one hasn’t already been defined. Then, if you can think of a common function that operates in a similar way to the function you're trying to interface, start by copying its definition and modifying it. In either case, you must understand the way that the interfaces work, and to do this, you must first understand their goal.

The `malloc` function returns blocks of memory, and we need to tell Insure++ about the size and location of such blocks. This is the reason for the call to `iic_alloc` at line 9 in Figure 13. This is the interface function that tells Insure++ to record the fact that a block of uninitialized memory of the given size has been allocated. From then on, references to this block of memory will be understood properly by Insure++.

Similarly, the purpose of `memcpy` is to take a number of bytes from one particular location and copy them to another. This activity is indicated by the call to the interface function `iic_copy` at line 32 of Figure 13. Insure++ uses this call to understand that two memory regions of the indicated size will be read and written, respectively.

The other code shown in the interface descriptions is used to check that parameters lie in legal ranges and is used to provide additional error checking.

Using interfaces

To use an interface, we first compile it with the Insure++ interface compiler, `iic`. If, for example, we put the interface for the `lib_gimme` function, shown on page 104, in a file called `gimme_i.c`, we would use the command

```
iic gimme_i.c
```

This results in the file `gimme_i.tqs`, which can be passed to Insure++ on the command line as follows:

```
insure -c gimme_i.tqs wilduse.c
insure -o wild wilduse.o mylib.a
```

in which we assume that the library containing the actual code for the `lib_gimme` routine is called `mylib.a`.

An additional example of how to use an interface can be found earlier in this section on page 107.

The basics for using an interface, therefore, are to:

- Compile the interface with `iic`.
- Recompile your program.

Note that you don't have to limit yourself to a single interface per source file. If you are preparing an interface module for an entire library, or a source file with multiple functions, you can put them all into the same interface description file.

Similarly, you don't have to pass all the names of your compiled interface modules on the `insure` command line every time. You can add lines to your `.psrc` files that list interface modules as follows

```
insure++.interface_library /usr/trf/mylib.tqs
insure++.interface_library /usr/local/ourlib.tqs
```

Ordering of interfaces

Files containing compiled interface definitions can be placed in any directory. Insure++ can be told to use such files in various ways, and processes them according to the following rules:

- If a standard library interface exists it is processed first.
- Interfaces specified in `interface_library` statements in configuration (`.psrc`) files are processed next, potentially overriding standard library definitions.
- Interface modules (i.e., files with the suffix `.tqs` or `.tqi`) specified on the `insure` command line override any other definitions.

Later definitions supercede earlier ones, so you can make a local definition of a library function and it will override the standard one in the library.

To see which interface files will be processed, and in which order, you can execute the command

```
iiwhich -l
```

which lists all the standard library files for your system, and then any indicated by `interface_library` commands in configuration files.

To find a function in an interface library, you can use the `iiwhich` command as already described. To list the contents of a particular .tqs, .tqi, or .tql file, use the `iiinfo` command.

Working on multiple platforms or with multiple compilers

Many projects involve porting applications to several different platforms or the use of more than one compiler. Insure++ deals with this by using two built-in variables, which denote the machine architecture on which you are running and the name of the compiler you are using. You can use these values to switch between various combinations, each specific to a particular machine or compiler.

For example, environment variables, '~'s (for `HOME` directories) and the '%' notation described on page 134, are expanded when processing filenames, so the command

```
insure++.interface_library
$HOME/insure++/%a/%c/foo.tqs
```

loads an interface file with a name such as

```
/usr/me/insure++/sun4/cc/foo.tqs
```

in which the environment variable `HOME` has been replaced by its value and the ‘`%a`’ and ‘`%c`’ macros have been expanded to indicate the architecture and compiler name in use. This allows you to load the appropriate .tqs, .tqi, or .tql files for the architecture and compiler that you are using.

One problem to watch out for occurs when you switch to a compiler for which Insure++ supplies no interface modules. In this case, you will see an error message during compilation. Several work-arounds are possible as described in the FAQ (`FAQ.txt`).

Common interface functions

Most definitions need only a handful of interface functions of which we’ve already introduced the most common:

```
void iic_alloc(void *ptr, unsigned long size);
```

Declares a block of dynamically allocated, uninitialized, memory.

```
void iic_source(void *ptr, unsigned long size);
```

Declares that a block of memory is read.

```
void iic_sourcei(void *ptr, unsigned long size);
```

Declares that a block of memory is read and also checks that it is completely initialized.

```
void iic_dest(void *ptr, unsigned long size);
```

Declares that a block of memory is modified.

```
void iic_copy(void *to, void *from,
                  unsigned long size);
```

Declares that the indicated block of memory is copied.

```
void iic_error(int code, char *format, ...);
```

Causes an error to be generated with the indicated error code.

Subsequent arguments are treated as though they were part of the `printf` statement.

Other commonly occurring functions are listed below together with examples of system calls that use them. You can use the `iiwhich` command on the listed functions to see examples of their use.

`int iic_string(char *ptr, unsigned long size);`

Declares that the argument should be a `NULL` terminated character string. This is used in most of the string handling routines such as `strcpy`, `strcat`, etc. The second argument is optional, and can be used to limit the check to at most `size` characters.

`void iic_alloci(void *ptr, unsigned long size);`

Declares a block of dynamically allocated, initialized memory such as might be returned by `calloc`.

`void iic_allocs(void *ptr, unsigned long size);`

Declares a pointer to a block of statically allocated memory. Used by functions that return pointers into static strings. `ctime` and `getenv` are examples of system calls that do this.

`void iic_unalloc(void *ptr);`

Declares that the indicated pointer is de-allocated with the system call `free`.

A complete list of available functions is given in “Interface Functions” on page 381.

Checking for errors in system calls

We can make interfaces even more user-friendly by adding checks for common problems, similar to the user level checks that were described in “Code Insertions” on page 97.

For example, `malloc` can fail. This is the reason for the second branch of the code in line 11 of Figure 13. If the actual call to `malloc` fails, instead of telling Insure++ about a block of allocated memory with `iic_alloc`, we cause an

Insure++ error with code RETURN_FAILURE and the error message shown. This, in turn, will cause a message to be printed (at runtime) whenever malloc fails and the RETURN_FAILURE error code has been unsuppressed. (See “Enabling error messages” on page 72.)

Similarly, memcpy can cause undefined behavior when given perfectly valid buffers that happen to overlap. We check for this case in the code at line 20, and again, cause an Insure++ error if a problem is detected.

This method provides a very powerful debugging technique, which is used extensively in the interface files supplied with Insure++. Since the RETURN_FAILURE error code is suppressed by default, you will normally not be bothered by messages when system calls fail. The assumption is that the user application is going to deal with the problem. In fact, it may require certain system calls to fail in order to work properly. However, when particularly nasty bugs appear, it is often useful to enable the RETURN_FAILURE error category to look for cases where system calls fail “unexpectedly” and are not being handled correctly by the application. Errors such as missing files (causing fopen to fail) or insufficient memory (malloc fails) can then be diagnosed trivially.

Using Insure++ in production code

A particularly powerful application of the technique described in the previous section is to make two different versions of your application:

- One with full error checking
- One without Insure++ at all

The first of these is used during application development to find the most serious bugs. The second is the one that will be used in production and shipped to customers.

When you or your customer support team is faced with a problem, they can run this code with the RETURN_FAILURE error class enabled and look for “unexpected” failures such as missing files, incorrectly set permissions, insufficient memory, etc.

Advanced interfaces: complex data types

The interfaces that have been considered so far are simple in the sense that their behavior is determined by their arguments in a straightforward manner.

To show a more complex example, consider the following data structure

```
struct mybuf {
      int len;
      char *data;
};
```

This data type could be used to handle variable length buffers. The first element shows the buffer length and the second points to a dynamically allocated buffer.

The code which allocates such an object might look as follows:

```
#include <stdlib.h>

struct mybuf *mybuf_creat(n)
      int n;
{
      struct mybuf *b;

      b = (struct mybuf *)malloc(sizeof(*b));
      if(b) {
          b->data = (char *)malloc(n);
          if(b->data) b->len = n;
          else b->len = 0;
      }
      return b;
}
```

Similarly, we might define operations on a `struct mybuf` that work in quite complex ways on its data.

To build an interface description of the `mybuf_create` function which detailed all its behavior would require the following code

```
struct mybuf *mybuf_creat(n)
      int n;
{
      struct mybuf *b;

      b = mybuf_creat(n);
      if(b) {
           iic_alloci(b, sizeof(*b));
           if(b->data)
                iic_alloc(b->data, b->len);
      }
      return b;
}
```

Note how the structure of the interface description follows that of the original source.

This matching would be seen in the interface descriptions of all the other functions that operate on the `struct mybuf` data type, too. In fact, the interface description would probably end up looking quite a lot like the source code!

There are three standard approaches to dealing with this problem:

- Forget the interface entirely, process the real source code with `insure,` and link it in the normal manner.
- "Go deep" and define an interface that mimics all of the details of the interface, including all the operations on the internal structure elements.
- "Go opaque" and build an interface that defines some levels of the functions without necessarily going into details of their action.

Each of these is a good approach in a different situation.

The first approach is the best in terms of accuracy and reliability. Given the original source code, Insure++ will have complete knowledge of the workings of the code and will be able to check every detail itself.

The second approach is best when the source code is unavailable but you still want to check every detail of your program's interaction with the affected routines. It can be implemented only if you have intimate knowledge of how the routines work, since you will have to use the interface functions to mimic the actions of the functions on the individual elements of the `struct mybuf`.

The third approach is appropriate when you are sure that the functions themselves work correctly. Perhaps, for example, you've been running Insure++ on their source code at some earlier date and you know that they are internally consistent and robust. In this case, you may want to increase the performance of the rest of your program by checking the high level interface to the routines, but not their internal details.

Another reason for adopting the third approach might be that you actually don't know the details of the functions involved and might not be able to duplicate their exact behavior. A good example would be building an interface to a third party library. You have clear definitions of the upper level behavior of the routines, but may not know how they work internally.

The first and second approaches have already been discussed. The third approach is easily achieved by doing nothing - Insure++ will recognize that the data type has not been declared in detail and should therefore not be checked in detail. You can choose for yourself which fields to declare in detail and which to ignore.

Interface esoterica

Since it is possible to express a wide range of actions in C, interface files must have correspondingly sophisticated capabilities in order to define their actions and check their validity.

One of these features was seen in the previous section: the `iic_startup` function. This function can be defined in any interface file and contains calls to interface functions that will be made before calling *any* of the other functions defined in the interface file. Typically, you will place definitions and initializations of known global or external variables in this function.

Note that each interface file may have its own iic_startup.

Variable argument lists are dealt with by using the pre-defined variable __dots__. For example, the interface specification for the standard system call printf is

```
int printf(char *format, ...)
{
      iic_string(format);
      iic_output_format(format);
      return(printf(format, __dots__));
}
```

The variable __dots__ in the function call matches the variable arguments declared with "..." in the definition.

Checking of printf and scanf style format strings is done with the iic_output_format and iic_input_format routines. These check that arguments match their corresponding format characters. iic_strlenf returns the length of a string after its format characters have been expanded and can be used to check that buffers are large enough to hold formatted strings.

A complete list of interface functions can be found in "Interface Functions" on page 381.

Callbacks

In many programming styles, such as programming in the X Window System or when using signal handlers, functions are registered and are then "called-back" by the system. Often the user program contains no explicit calls to these functions.

If the callback functions use *only* variables that are defined in the user program, nothing unusual will happen, since Insure++ will understand where all this data came from and will keep track of it properly. In many cases, however, the library function making the callback will pass additional data to the called function that was allocated internally, which Insure++ never saw.

For example:

- UNIX functions such as `qsort` and `scandir` take function pointer arguments that are called back from within the system function.
- Signal handling functions often pass to their handlers a data structure containing hardware registers and status information.
- The X Window System library often passes information about the display, screen, and/or event type to its callback functions.

In these cases, Insure++ will attempt to look up information about these data structures without finding any, which limits its ability to perform strong error checking.

This is not a serious limitation - it merely means that the unknown variables will not be checked as thoroughly as those whose allocation was processed by Insure++.

If you wish to improve the checking performed by Insure++ in these cases, you can use the interface technology in two different ways:

- You can make interfaces to the functions that install or register the callbacks (with `iic_callback`) indicating how to process their arguments when the callbacks are invoked.
- You can make interface definitions for your callback functions themselves, adding the keyword `iic_body` to their definition.

These two options are discussed in the next sections.

Using iic_callback

The first of these approaches is more general, since it allows you to define, in a single interface specification, the behavior of *any* callback that is installed by the function specified. To see how this works, consider the standard utility sorting function, qsort. One of the arguments to this routine is a function pointer that is used to compare pairs of elements during sorting.

The following interface to this function checks that the qsort function does no more than N^2 comparisons, where N is the number of elements (this may or may not be a sensible check, but serves the purpose of explaining callback interfaces):

```
#include <sys/stdtypes.h>
#include <math.h>

static int _qsort_num_comparisons;

static int _qsort_cb(void *e1, void *e2)
{
     _qsort_num_comparisons += 1;
     return _qsort_cb(e1, e2);
}

void qsort(void *base, size_t nelem,
     size_t width,
     int (*func)(void *, void *))
{
     iic_dest(base, nelem*width);
     iic_func(func);
     iic_callback(func, _qsort_cb);
     _qsort_num_comparisons = 0;
     qsort(base, nelem, width, func);
     if (_qsort_num_comparisons >
                        nelem * nelem)
          iic_error(USER_ERROR,
               ”Qsort took %d compares.”,
               _qsort_num_comparisons);
}
```

The main body of the interface is in lines 16-25.

Line 16 checks that the pointer supplied by the user indicates a large enough region to hold all the data to be sorted, while line 17 checks that the function pointer actually points to a valid function. Line 20 contains the call to the normal `qsort` function.

The interesting part of the interface is the call to `iic_callback` in line 18. The two arguments connect a function pointer and a “template”, which in interface terms is the name of a previously declared static function; in this case `_qsort_cb`, declared in lines 6-10. The template tells Insure++ what to do whenever the system invokes the called-back, user-supplied function. In this particular case, the interface merely increments a counter so we can see how many times the callback gets called (note that we set the counter to 0 on line 19 of the `qsort` interface). In general, you can make any other interesting checks here before or after invoking the callback function.

Notice that once this interface is in use, it automatically processes *any* function that gets passed to the `qsort` function.

Using `iic_body`

The second callback option is to define interfaces for each individual function that will be used as a callback.

Consider, for example, the X Window System function `XtAddCallback`, which specifies a function to be called in response to a particular user interaction with a user interface object. It is quite common for code to contain many calls to this function, for example

```
XtAddCallback(widget, ..., myfunc1, ...);
XtAddCallback(widget, ..., myfunc2, ...);
XtAddCallback(widget, ..., myfunc3, ...).
```

One solution for this routine would be to provide an `iic_callback` style interface for the `XtAddCallback` function as described in the previous section.

The second method is to specify interfaces to the called-back functions themselves, with the additional iic_body keyword. An interface for the routine myfunc1 might be written as follows:

```
/*
 * Interface definition for callback function
 * uses the iic_body keyword.
 */
void iic_body myfunc1(Widget w,
      XtPointer client_data,
      XtPointer call_data)
{
      if (!call_data)
          iic_error(USER_ERROR,
              “myfunc1 passed NULL call_data”);
      myfunc1(w, client_data, call_data);
      return;
}
```

This interface checks that myfunc1 is never passed NULL client_data.

Note that in this scenario you would have to specify three separate interfaces; one each for myfunc1, myfunc2 and myfunc3. (And, indeed, any other functions used as callbacks.)

Which to use: iic_callback or iic_body?

From the previous discussion it might seem that iic_callback should always be preferred over iic_body, since it is more general and less code must be written. Unfortunately, the general iic_callback method has a severe limitation: the code generated by Insure++ when you use iic_callback is good for “immediate use only”.

To understand what this means, consider the difference between the two cases already discussed:

- In the qsort example, the iic_callback function made the association between function pointer and template, which was then immediately used by the qsort function. By the time the interface code returns to its caller, the connection between function and template is no longer required.
- In the X Window System example, the callbacks registered by the XtAddCallback function are expected to survive for the remainder of the application (or until cancelled by another X Window System call). Similarly, the connection between function pointer and template is expected to survive as long.

As a consequence, the iic_callback method is only applicable to a small number of circumstances, and in general you must either:

- Use the iic_body method
- Do nothing, and allow Insure++ to skip checks on unknown arguments to callback functions.

Conclusions

Interfaces play an important, but optional, role in the workings of Insure++.

If you wish, you can always eliminate error messages about library calls by adding suppress options to your .psrc files and running your program again. This approach has the advantage of being very quick and easy to implement, but discards a lot of information about your program that could potentially help you find errors.

To capture all the problems in your program, you need to use interfaces. Insure++ is supplied with interfaces for all the common functions and quite a few uncommon ones. These are provided in source code form in the directory src.$ARCH/$COMPILER so that you can look at them and modify them for your particular needs.

The iiwhich command can help you find existing definitions that can then be used as building blocks in making your own interfaces.

If you build an interface to a library that you'd like to share with other users of Insure++, please send it to us (insure@parasoft.com) and we'll make it available.

Part II

Insure++
Reference Guide

Configuration Files

Insure++ programs read options from files called .psrc, which may exist at various locations in the file system. These options control the behavior of Insure++ and programs compiled with Insure++. The files are processed in the order specified below.

Earlier versions of Insure++ used configuration files called .insight. These files are still supported by this version, but will not be in the next version. Any of the options on the following pages can also be used in .insight files, but without the "insure++." prefix. However, we recommend that users move to the newer .psrc files as soon as they can.

- The file .psrc in the appropriate lib and compiler subdirectories of the main Insure++ installation directory, e.g.

  ```
  /usr/local/insure/lib.sol/cc/.psrc
  ```

  or

  ```
  /usr/local/insure/lib.rs6000/xlC/.psrc
  ```

- The file .psrc in the main installation directory.
- A file .psrc in your $HOME directory, if it exists.
- A file .psrc in the current working directory, if it exists.
- Files specified with the -Zop switch and individual options specified with the -Zoi switch to the insure command in the order present on the command line.

In each case, options found in later files override those seen earlier. All files mentioned above will be processed and the options set before any source files are processed. You can also override these options at runtime by using the _Insure_set_option function.

Typically, compiler-dependent options are stored in the first location, site-dependent options are stored in the second location, user-dependent options are stored in the third location, and project-dependent options are stored in the fourth location. `-Zop` is commonly used for file-dependent options, and `-Zoi` is commonly used for temporary options.

Format

Insure++ configuration files are simple ASCII files created and modified with a normal text editor.

Entries that begin with the character ‘#’ are treated as comments and ignored, as are blank lines.

All keywords can be specified in either upper or lower case, and embedded underbar characters (‘_’) are ignored. Arguments can normally be entered in either case, except where this has specific meaning, such as in directory or file names.

If a line is too long, or would look better on multiple lines, you can use the ‘\’ character as a continuation line.

Working on multiple platforms or with multiple compilers

Many projects involve porting applications to several different platforms or the use of more than one compiler. Insure++ deals with this issue by using two built-in variables, which denote the machine architecture on which you are running and the name of the compiler you are using. Anywhere that you would normally specify a pathname or filename, you can then use these values to switch between various options, each specific to a particular machine or compiler.

For example, environment variables, ‘~’s (for HOME directories) and the ‘%’ notation described on page 134 are expanded when processing filenames, so the command

```
interface_library $HOME/insure/%a/%c/foo.tqs
```

loads an interface file with a name such as

```
/usr/me/insure/sun4/cc/foo.tqs
```

in which the environment variable HOME has been replaced by its value and the ‘%a’ and ‘%c’ macros have been expanded to indicate the architecture and compiler name in use.

There is one additional comment that must be made here. In the compiler-default .psrc files, there are several interface_library options of the form

```
Insure++.InterfaceLibrary
$PARASOFT/lib.%a/%c/builtin.tqi \
$PARASOFT/lib.%a/libtqsiic%c.a
```

Despite appearances, the PARASOFT used above is not a true environment variable. If the PARASOFT environment variable is not set by the user, it will be expanded automatically by Insure++ itself.

Option values

The following sections describe the interpretation of the various parameters. They are divided into two classes: compile time and runtime. Modifying one of the former options requires that files be recompiled before it can take effect. The latter class merely requires that the program be executed again.

Some options have default values, which are printed in the following section in **boldface**.

Filenames

A number of the Insure++ options can specify filenames for various configuration and/or output files. You may either enter a simple filename or give a template which takes the form of a string of characters with tokens such as “`%d`”, “`%p`”, or “`%V`” embedded in it. Each of these is expanded to indicate a certain property of your program as indicated in the following tables. The first table lists the options that can be used at both compile and runtime.

Key	Meaning
`%a`	Machine architecture on which you are running, e.g., `solaris`, `aix4`, `hp10`, etc.
`%c`	Abbreviated name of the compiler you are using, e.g. `cc`, `gcc`, `5cc`, etc.
`%r`	Insure++ version number, e.g. `5.2`
`%R`	Insure++ version number without ‘`.`’ characters, e.g., version 5.2 becomes `52`
`%t`	`.tqs` file format version number, e.g., `5.2.0`
`%T`	Similar to ‘`%t`’ but with ‘`.`’ characters removed

This second table lists the tokens available only at runtime.

Key	Meaning
`%d`	Time of program compilation in format: `YYYYMMDDHHMMSS`
`%D`	Time of program execution in format: `YYYYMMDDHHMMSS`
`%n`	Integer sufficient to make filename unique, starting at 0
`%p`	Process I.D.
`%v`	Name of executable
`%V`	Directory containing executable

Thus, the name template

```
insure++.report_file %v-errs.%D
```

when executed with a program called `foo` at 10:30 a.m. on the 21st of March 2001, might generate a report file with the name

```
foo-errs.20010321103032
```

(The last two digits are the seconds after 10:30 at which execution began.)

You can also include environment variables in these filenames so that

```
$HOME/reports/%v-errs.%D
```

generates the same filename as the previous example, but also ensures that the output is placed in the `reports` sub-directory of the user’s `HOME`.

Options at runtime and compile time

Several of the Insure++ options have effects during both compilation and program execution. When the option is active, it is controlled by an extra qualifier keyword as shown in the examples below.

The option

```
insure++.runtime.suppress READ_NULL
```

suppresses errors in the `READ_NULL` class during program execution. An error in this class detected during compilation would still be reported.

Similarly,

```
insure++.compile.unsuppress BAD_PARM(sign)
```

enables the display of this error category during compilation, but not during program execution. Compile time options also apply at link time.

If you wish to apply the same option to both compilation and execution, simply omit the qualifier.

```
insure++.suppress EXPR_NULL
```

Using -Zop and -Zoi

On the command line, the -Zop and -Zoi options are processed from left to right, *after* all other .psrc files, and *before* processing any source code. Therefore, the following command line would tell Insure++ to compile all the files with the cc compiler.

```
insure -Zoi “compiler gcc” -o foo foo.c
      -Zop foo.def foo2.c -Zoi “compiler g++”
      foo3.c -Zoi “compiler cc”

foo.def:
compiler CC
```

Compiled-in options

Insure++ now encodes certain options at compile-time into the actual binary that is built. Basically, Insure++ uses the information available at compile time, e.g. compiler name or executable name, and encodes a .psrc option into the binary itself. This option can be overridden in a .psrc file using the ! character. For example, if you build your executable with one name and run it in another directory or with a different name, you could use an option like

```
!exename /home/user/tmp/bar
```

to override the option inside the binary.

Options used by Insure++

Compiling/linking

`insure++.auto_expand [all|off|`**`on`**`]`

Specifies how Insure++ should treat suspected “stretchy” arrays. See ”Report summaries” on page 72 for a discussion of stretchy arrays, and the table below for explanations of the allowed keywords for this option. Multi-dimensional arrays are never automatically expanded. To tell Insure++ that a specific array is stretchy, use the `expand` option (see page 145).

Keyword	Meaning
`all`	All arrays at the end of structs, classes, and unions are treated as stretchy, regardless of size
`off`	No automatic detection of stretchy arrays
`on`	If the last field of a struct, class, or union is an array and has no size, size 0, or size 1, it is treated as stretchy. Note that only some compilers allow 0 or empty sizes, but size 1 is very common for stretchy arrays

`insure++.c_as_cpp [on|`**`off`**`]`

C++

Specifies whether files with the `.c` extension should be treated as C++ source code. With this option `off`, Insure++ will treat files with the `.c` extension as C code only. If you use C++ code in `.c` files, you should turn this option `on`.

`insure++.checking_level [`**full**`|minimal]`

Specifies **do_mpf** level of instrumentation. **do_mpf** is the code insert program for compiling with Insure++.

`insure++.checking_uninit [`**`on`**`|off]`

On means insert code to check for uninitialized variables. Off means do not insert code to check for uninitialized variables. If turned off, runtime uninitialized variable checking is then limited to uninitialized pointer variables (See page 19). See page 153 for the runtime effects of this option.

`insure++.compile.reportfile [filename|insra|`**`stderr`**`]`

Specifies the name of the report file for compile time messages only. Environment variables and various pattern generation keys may appear in **filename**. (See Filenames on page 134.) Use of the special filename **Insra** tells Insure++ to send its output to Insra.

`insure++.compiler compiler_name`

Specifies the name of an alternative compiler, such as `gcc`. If your new compiler is not recognized by Insure++, you may have to set the `compiler_acronym` option. This option overrides all other `compiler_*` options: `compiler_acronym`, `compiler_c`, `compiler_cpp`, and `compiler_default`. The indicated compiler will be called every time `insure++` is called.

`insure++.compiler_acronym abbreviation`

Specifies the colloquial name of an alternative compiler, such as `gcc`. This name is used to locate the appropriate `.psrc` files and `TQS` library modules. It does not indicate which compiler will actually be called to compile source files (see the other `compiler_*` options). This option overrides the `compiler_c`, `compiler_cpp`, and `compiler_default` options. In addition, this option must be placed *after* the active `compiler` option. The order must be

```
compiler c89
compiler_acronym cc
```

and not vice versa.

`insure++.compiler_c C_compiler_name`

Specifies the name of the default C compiler, such as `gcc`. This compiler will be called for any `.c` files. The default is `cc`. This option is overridden by the `compiler` and `compiler_acronym` options.

`insure++.compiler_cpp C++_compiler_name`

Specifies the name of the default C++ compiler, such as `g++`. This compiler will be called for any `.cc`, `.cpp`, `.cxx`, and `.C` files. The default is platform dependent: `cxx` for Alpha, `g++` for Linux, `xlC` for RS/6000, and `CC` for other platforms. This option is overridden by the `compiler` and `compiler_acronym` options.

`insure++.compiler_default [`**`c`**`|cpp]`

Specifies whether the default C or C++ compiler should be called to link when there are no source files on the link line. This option is overridden by the `compiler` and `compiler_acronym` options.

```
insure++.compiler_deficient
          [all|address|cast|enum|member_pointer|
          scope_resolution|static_temps|
          struct_offset|types|no_address|no_cast|
          no_enum|no_member_pointer|
          no_scope_resolution|no_static_temps|
          no_struct_offset|no_types|none]
```

Specifies which features are not supported by your compiler. The default is compiler-dependent.

Keyword	Meaning
`all`	Includes all positive keywords
`address/no_address`	
`cast/no_cast`	
`enum/no_enum`	

Keyword	Meaning
member_pointer/no_member_pointer	
scope_resolution/no_scope_resolution	
static_temps/no_static_temps	
struct_offset/no_struct_offset	
types/no_types	
none	Compiler handles all cases

Different compilers require different levels of this option as indicated in the compiler-specific README files and in ($PARASOFT)/lib.$ARCH/$compiler.

insure++.compiler_fault_recovery [off|**on**]

This option controls how Insure++ recovers from errors during compilation and linking. With fault recovery on, if there is an error during compilation, Insure++ will simply compile with the compiler only and will not process that file. If there is an error during linking, Insure++ will attempt to take corrective action by using the -Zsl option. If this option is turned off, Insure++ will make only one attempt at each compile and link.

insure++.compiler_keyword
[*|const|inline|signed|volatile] keyword

Specifies a new compiler keyword (by using the *) or a different name for a standard keyword. For example, if your compiler uses __const as a keyword, use the option

```
compiler_keyword const __const
```

`insure++.compiler_options keyword value`

Specifies various capabilities of the compiler in use, as described in the following table.

Keyword	Value	Meaning
`ansi`	None	Assumes compiler supports ANSI C (default)
`bfunc <type>`	Function name	Specifies that the given function is a "built-in" that is treated specially by the compiler. The optional `type` keyword specifies that the built-in has a return type other than `int`. Currently, only `long`, `double`, `char *`, and `void *` types are supported.
`btype`	Type name	Specifies that the given type is a "built-in" that is treated specially by the compiler
`bvar <type>`	Variable name	Specifies that the given variable is a "built-in" that is treated specially by the compiler. The optional `type` keyword specifies that the built-in has a return type other than `int`. Currently, only `long`, `double`, `char *`, and `void *` types are supported.
`esc_x`	Integer	Specifies how the compiler treats the '`\x`' escape sequence. Possible values are 0 treat '`\x`' as the single character '`x`' (Kernighan & Ritchie style) -1 treat as a hex constant. Consume as many hex digits as possible >0 treat as a hex constant. Consume at most the given number of hex digits

Keyword	Value	Meaning

`for_scope`	`nested`
	`notnested`
	`optional`
		Specifies how `for(int i; ...; ...)` is scoped. Possible values are
		nested — New ANSI standard, always treat as nested.
		notnested — Old standard, never treat as nested.
		optional — New standard by default, but old-style code is detected and treated properly (and silently)

`knr`	None	Assumes compiler uses Kernighan and Ritchie (old-style) C
`loose`	None	Enables non-ANSI extensions (default)
`namespaces`	None	Specifies that `namespace` is a keyword (default)
`nonamespaces`	None	Specifies that `namespace` is not a keyword
`promote_long`	None	Specifies that integral data types are promoted to `long` in expressions, rather than `int`
`sizet`	`d`, `ld`, `u`, `lu`	Specifies the data type returned by the `sizeof` operator, as follows: `d`=`int`, `ld`=`long`, `u`=`unsigned int`, `lu`=`unsigned long`.
`strict`	None	Disables non-ANSI extensions (compiler dependent)
`xfunctype`	Function name	Indicates that the named function takes an argument which is a data type rather than a variable (e.g., `alignof`)

`insure++.coverage_boolean [`**`on`**`|off]`

Determines how Insure++ will report code execution. **On** tells Insure++ to keep track of whether each block was executed. **Off** tells Insure++ to count how many times each block is executed.

`insure++.coverage_map_data [`**`on`**`|off]`

The default for this option is **on.** Selecting **off** turns off coverage analysis used by the TCA add-on. This will make your Insure++-compiled programs smaller and faster.

`insure++.coverage_map_file [filename]`

Specifies the full path of the directory where Insure++ writes the `tca.map` file.

`insure++.demangle_method [external <filtname>|CC|gcc]`

Specifies compiler-specific algorithm for demangling function names. Currently supported compiler algorithms are C-Front based C++ compilers (e.g. CC). If you are using a different compiler, Insure++ understands most other demangling formats as well. The `filter <filtname>` option allows the use of the external demangler `filtname`. The default is compiler-dependent. See the compiler level .psrc file, which is in the directory lib.$ARCH/$COMPILER. This option is a compiled-in option, so you will need to prepend a **!** to the option in a `.psrc` file to change the setting at runtime. See Compiled-in options on page 137 for more details.

`insure++.disable_keyword [keyword]`

If you are using a built-in keyword as an identifier, this option tells Insure++ not to use this ANSI keyword as a keyword.
Example: `disable_keyword export`
This allows the user to use “export” as a variable or function name instead of a keyword.
Note that this option only disables built-in keywords, not keywords added with the `compilerkeyword.psrc` option.

insure++.dynamic_linking [**on**|off]

By default, Insure++ links its libraries dynamically or statically according to the current link options. Setting this option to off tells Insure++ not to allow its libraries to be linked dynamically, even though user and/or system libraries may still be linked dynamically. This option is the opposite of static_linking. (See page 151)

insure++.error_format string

Specifies the format for error message banners generated by Insure++. The string argument will be displayed as entered with the macro substitutions taking place as shown in the following table. The string may also contain standard C formatting characters, such as '\n'. (For examples, see page 62)

Key	Expands to
%c	Error category (and sub-category if required)
%d	Date on which the error occurs (DD-MON-YYYY)
%f	Filename containing the error
%F	Full pathname of the file containing the error
%h	Name of the host on which the application is running
%l	Line number containing the error
%p	Process ID of the process incurring the error
%t	Time at which the error occurred (HH:MM:SS)

insure++.expand subtypename

Specifies that the named structure element is "stretchy". See "Report summaries" on page 72 for a discussion of stretchy arrays. See also the auto_expand option on page 138 for details on automatic detection and handling of stretchy arrays.

`insure++.file_ignore string`

Specifies that any file which matches the string will not be processed by Insure++, but will be passed straight through to the compiler. The string should be a glob-style regular expression.

This option allows you to avoid processing files that you know are correct. This can significantly speed up execution and shrink your code.

`insure++.function_ignore file::function_name`

This option tells Insure++ not to instrument the given function (the file qualifier is optional). This is equivalent to turning off the checking for that routine. If the function in question is a bottle-neck, this may dramatically increase the runtime performance of the code processed with Insure++.

`function_name` can now (version 3.1 and higher) accept the `*` wildcard. For example, the option

```
insure++.function_ignore foo*
```

turns off instrumentation for the functions `foo`, `foobar`, etc.

`insure++.header_ignore string`

Specifies that any function in the filename specified by the string will not be instrumented by Insure++. The string should be a glob-style regular expression and should include the full path.

This option allows you to avoid doing runtime checking in header files that you know are correct. This can significantly speed up execution and shrink your code. Please note, however, that the file must still be parsed by Insure++, so this option will not eliminate compile-time warnings and errors, only runtime checking.

insure++.interface_defaults [all|*|alloc|new|
delete|**::alloc**|::new|::delete|
off|none]

Specifies for which functions to use standard interfaces. The standard interfaces assume that new and delete behave like the global new and delete in allocating a block of memory. If your functions adhere to those guidelines, you can specify use of the standard interfaces with this option.

Keyword	Meaning
all/*	Includes alloc and ::alloc
alloc	Includes new and delete
new	Assume all member news are “standard”
delete	Assume all member deletes are “standard”
::alloc	Includes ::new and ::delete
::new	Assume global new is “standard”
::delete	Assume global delete is “standard”
off/none	Assume nothing about new and delete

`insure++.interface_disable key`

This option tells Insure++ not to use the interfaces specified by the `key`. The interfaces will then not be inserted during instrumentation at compile-time. The `key` can be obtained by looking in the compiler default `.psrc` file (see page 131) for the `interface_library` option specifying the interfaces you wish to disable and removing the `lib` prefix and the `.tqi` file extension. For example, to turn off the C library interfaces, use the option

```
interface_disable c
```

`insure++.interface_enable key`

This option tells Insure++ to use the interfaces specified by the `key`. The interfaces will then be inserted during instrumentation at compile-time. The `key` can be obtained by looking in the compiler default `.psrc` file (see page 131) for the `interface_library` option specifying the interfaces you wish to enable and removing the `lib` prefix and the `.tqi` file extension. For example, to turn on the C library interfaces, use the option

```
interface_enable c
```

`insure++.interface_ignore function_name`

This option tells Insure++ not to use its interface for `function_name`.

`insure++.interface_library file1, file2, file3, …`

Specifies Insure++ interface modules to be used on each compile. Equivalent to specifying the list on the `insure` command line. Filenames may include environment variables and Insure++ macros to help cross-platform development as described on page 132.

`insure++.interface_preference [tq1|tqi|tqs]`

Specifies which interfaces are to be used in the case of conflicts. Interface conflicts (two interfaces for the same function but of different types) will be resolved in favor of the type occurring first in the interface preference list.

Types are:
tq1--LRT interfaces
tqi--object interfaces (the Insight_XXX versions)
tqs--inline interfaces

Examples:

`interface_preference tql tqi tqs` (The default) Specifies preference in the following order: Use LRT first; if LRT is not available then object interfaces; if this is not available then use inline interfaces. On UNIX platforms, tqi is the default interface. The tql interface is only enabled when the customer specifies it, or when minimal instrumentation mode is running or the thread is on.

`interface_preference tqi tqs` (turns off LRT interfaces) Specifies preference in the following order: Prefer object interfaces to inline interfaces; do not use LRT interfaces.

`insure++.interface_reset`

Turns off all interfaces up to this point in the `.psrc` file. Additional `interface_library` options can be used after this line to add back certain interfaces.

`insure++.interface_statics [`**`on`**`|off]`

This option controls interface checking on static functions. If you want to have static functions with the same names as functions for which there are interfaces (e.g. `write`), you can turn off interface checking for these static functions by setting this option to `off`.

`insure++.linker linker_name`

Specifies the name of an alternative linker. This only applies if you are using the `ins_ld` command.

`insure++.linker_source source_code`

This option tells Insure++ to add the given code to its initialization file. This can help eliminate unresolved symbols caused by linker bugs.

`insure++.linker_stub symbol_name`

This option tells Insure++ to create and link in a dummy function for the given `symbol_name`. This can help eliminate unresolved symbols caused by linker bugs.

`insure++.malloc_replace [`**`on`**`|off]`

If `on`, Insure++ links its own version of the dynamic memory allocation libraries. This gives Insure++ additional error detection abilities, but may have different properties than the native library (for example, it will probably use more memory per block allocated). Setting this option to `off` links in the standard library. This option does not apply in LRT mode.

`insure++.pragma_ignore string`

Any pragma which matches the string will be deleted by Insure++. The string should be a glob-style regular expression.

`insure++.rename_files [on|`**`off`**`]`

Normally, Insure++ creates an intermediate file which is passed to the compiler. In some cases, this may confuse debuggers. If this is the case, you can set this option Insure++ will then rename the files during compilation so that they are the same. In this case, an original source file called `foo.c` would be renamed `foo.c.ins_orig` for the duration of the call to Insure++.

`insure++.report_banner [`**`on`**`|off]`

Controls whether or not a message is displayed on your terminal, reminding you that error messages have been redirected to a file. (See page 60)

`insure++.report_file [filename|insra|`**`stderr`**`]`

Specifies the name of the report file. Environment variables and various pattern generation keys may appear in `filename`. (See page 135) Use of the special filename `Insra` tells Insure++ to send its output to Insra.

`insure++.sizeof type value`

This option allows you to specify data type sizes that differ from the host machine, which is often necessary for cross compilation. `value` should be the number `sizeof(type)` would return on the target machine. Allowed `type` arguments are `char`, `double`, `float`, `int`, `long`, `long double`, `long long`, `short`, and `void *`.

`insure++.stack_internal [on|`**`off`**`]`

If you are using the `symbol_table off` runtime option (see page 164), you can set this option to `on` and recompile your program to get filenames and line numbers in stack traces without using the symbol table reader.

`insure++.static_linking [on|`**`off`**`]`

By default, Insure++ links its libraries dynamically or statically according to the current link options. Setting this option to `on` forces Insure++'s libraries to be linked statically, even though user and/or system libraries may still be linked dynamically. This option is the opposite of `dynamic_linking`.
(See page 145)

`insure++.stdlib_replace [`**`on`**`|off]`

Links with an extra Insure++ library that checks common function calls without requiring recompilation. This is useful for finding bugs in third-party libraries or for quickly checking your program without fully recompiling with Insure++. This option does not apply in LRT mode.

`insure++.suppress code`

Suppresses compile time messages matching the indicated error code. Context sensitive suppression does not apply at compile time (see page 67, *et seq*).

`insure++.suppress_output string`

Suppresses compile time messages including the indicated error string (see page 71). For example, to suppress the warning:

```
[foo.c:5] Warning: bad conversion in
assignment: char * = int *
 >> ptr = iptr;
```

add the following line to your `.psrc` file.

```
suppress_output bad conversion in
assignment
```

`insure++.suppress_warning code`

Suppresses C++-specific compile time messages matching the indicated warning code (see page 71). `code` should match the numerical code Insure++ prints along with the warning message you would like to suppress. The codes correspond to the chapter, section, and paragraph(s) of the draft ANSI standard on which the warning is based. For example, to suppress the warning:

```
Warning:12.3.2-5: return type may not be
specified for conversion functions
```

add the following line to your `.psrc` file.

```
suppress_warning 12.3.2-5
```

`insure++.temp_directory path`

Specifies the directory where Insure++ will write its temporary files, e.g. `/tmp`. The default is the current directory. Setting `path` to a directory local to your machine can dramatically improve compile-time performance if you are compiling on a remotely mounted file system.

`insure++.threaded_runtime [on|`**`off`**`]`

Specifies which Insure++ runtime library will be used at link time. This option should be turned on before linking threaded programs with Insure++.

`insure++.uninit_flow [1|2|3|...|`**`100`**`|...|1000]`

When Insure++ is checking for uninitialized memory, a lot of the checks can be deduced as either correct or incorrect at compile time. This value specifies how hard Insure++ should try to analyze this at compile time. A high number will make Insure++ run slower at compile time, but will produce a faster executable. Values over 1000 are not significant except for very complicated functions.

`insure++.unsuppress code`

Enables compile time messages matching the indicated error code. Context-sensitive suppression is not supported at compile time (see page 72, *et seq*).

`insure++.virtual_checking [`**`on`**`|off]`

Specifies whether VIRTUAL_BAD error messages will be generated. See page 354 for more information about this error message.

Running

`insure++.assert_ok filename::function`

Specifies that the return value of the given function (the file qualifier is optional) should be treated as an opaque object and not checked for errors (see page 72). This is primarily useful for eliminating errors reported in third-party libraries.

`insure++.autoreports [on|`**`off`**`]`

This boolean option is only accessed in twopass mode, which is, for all practical purposes, obsolete.

`insure++.checking_uninit [`**`on`**`|off]`

If set to `off`, this option specifies that the code to perform flow-analysis and checking for uninitialized variables should not be executed, if present. See page 139 for the compile time effects of this option. Runtime uninitialized variable checking is then limited to uninitialized pointer variables (see page 19).

`insure++.checking_uninit_min_size [1|`**`2`**`|3|...]`

Specifies the minimum size in bytes of data types on which Insure++ should perform full uninitialized memory checking. The default is `2`, which means that `char`s will not be checked by default. Setting this option to `1` will check `char`s, but may result in false errors being reported. These can be eliminated by using the `checking_uninit_pattern` option to change the pattern used (see below).

`insure++.checking_uninit_objects [on|off]`

Tells Insure++ to check for uninitialized objects. This option inserts into the constructor (if one exists) and makes calls to write 0xdeadbeef.

`insure++.checking_uninit_pattern pattern`

Specifies the pattern to be used by the uninitialized memory checking algorithm. The default is `deadbeef`. `pattern` must be a valid, 8-digit hexadecimal value.

`insure++.checking_uninit_stack_scribble [on|`**`off`**`]`

Specifies extra uninitialized memory checking on the stack. This checking is not compatible with all compilers. If you get unusual core dumps after turning this option on, it is not compatible with your compiler and should be turned off.

`insure++.coverage_banner [on|`**`off`**`]`

Controls whether Insure++ displays a message about the coverage information that is written to `tca.log.`

`insure++.coverage_boolean [`**`on`**`|off]`

Determines how Insure++ will report code execution. **On** tells Insure++ to keep track of whether each block was executed. **Off** tells Insure++ to count how many times each block is executed.

`insure++.coverage_log_data [`**`on`**`|off]`

Specifies that Insure++ will write log data to the `tca.log` file.

`insure++.coverage_log_file [filename]`

Specifies the full path to the directory where Insure++ writes the `tca.log` file.

insure++.coverage_map_file [filename]

Specifies the complete path to the directory where Insure++ writes the `tca.map` file.

insure++.coverage_overwrite [**on**|off]

Determines if the `tca.log` file will be overwritten on each run.

insure++.coverage_switches [**on**|off]

Determines whether TCA will switch between coverage modes.

insure++.demangle [off|**on**|types|full_types]

Specifies the level of function name demangling in reports generated by Insure++. If you have a function

```
void func(const int)
```

you will get the following results:

Keyword	Result
off	func__FCi
on	func
types	func(int)
full_types	func(const int)

`!insure++.demangle_method [external <filtname>|CC|gcc]`

Specifies compiler-specific algorithm for demangling function names. Currently supported compiler algorithms are C-Front based C++ compilers (e.g. CC). If you are using a different compiler, Insure++ understands most other demangling formats as well. The `filter <filtname>` option allows the use of the external demangler `filtname`. The default is compiler-dependent. See the compiler level `.psrc` file, which is in the directory `lib.$ARCH/$COMPILER`.
This option is a compiled-in option, so you will need to prepend a `!` to the option in a `.psrc` file to change the setting at runtime. See page 137 for more details.

`insure++.error_format string`

Specifies the format for error message banners generated by Insure++. The string argument will be displayed as entered with the macro substitutions taking place as shown in the table on page 145. The string may also contain standard C formatting characters, such as ‘`\n`’. (For examples, see page 62.)

`!insure++.exename [<short_name>] filename`

Specifies the name of the executable, possibly with the path. This may be necessary to read the symbol table if Insure++ cannot find the executable. You will need to use the `!` character with this option to override the compiled-in `exename` option built into the binary at compile-time. (see page 137) There are two ways to use this option. The simplest is to omit the `short_name` argument and just specify the executable name with the path of the executable. For example, if you have compiled a file called `foo.c` into an executable named `foo` in the `/usr/local` directory, the correct `exename` option would look like:

```
!insure++.exename /usr/local/foo
```

Using the second option, you can specify the location of more than one executable. The `short_name` must be the name of executable *when it was linked*. The filename must be the full path and current name of the executable. For example, if you have built foo as above and also a program called bar (which was moved to your home directory and changed to foobar), you might use exename options like:

```
!insure++.exename foo /usr/local/foo
!insure++.exename bar ~/foobar
```

insure++.exit_hook [**on**|off]

Normally, Insure++ uses the appropriate atexit, onexit, or on_exit function call to perform special handling at exit. If for some reason, this is a problem on your system, you can disable this functionality via the exit_hook option.

insure++.exit_on_error [**0**|1|2|3|...]

Causes the user program to quit (with non-zero exit status) after reporting the given number of errors. The default is 0, which means that all errors will be reported and the program will terminate normally.

insure++.exit_on_error_banner [**on**|off]

Normally, when Insure++ causes your program to quit due to the exit_on_error option, it will print a brief message like the following:

```
** User selected maximum error count
reached: 10. Program exiting. **
```

Setting this option to off will disable this message.

insure++.free_delay [0|1|2|3|...|**119**|...]

This option controls how long the Insure++ runtime holds onto “free’d” blocks before allowing them to be reused. This is not necessary for error detection, but can be useful in modifying the behavior of your program for stress-testing. The number represents how many free’d blocks are held back at a time - large numbers limit memory reuse, and 0 maximizes memory reuse. Please note that this option is only active if malloc_replace was on during linking.

insure++.free_pattern pattern

Specifies a pattern that will be written on top of memory whenever it is freed. This pattern will be repeated for each byte in the freed region (this option is available only if `malloc_replace` was on at compile time). The default is `0`, which means no pattern will be written.

On some systems whose libraries assume freed memory is still valid, this may cause your program to crash.

insure++.free_trace [-1|0|1|2|3|...|**10**|...]

Specifies the maximum number of levels to track the stack whenever a block of dynamic memory is freed. Setting this option to a non-zero value tells Insure++ to include a description of the function call stack to at most the given depth whenever an error associated with the block is reported. Setting the value to `0` suppresses deallocation stack tracing, while the value `-1` traces the stack back to the `main` routine.

insure++.ignore_wild [on|**off**]

Specifies whether Insure++ will do checking for wild pointers. Turning this option `on` turns off wild pointer checking.

insure++.inuse [**on**|off]

When this option is enabled, it causes Insure++ to inform Inuse about all sbrk(2) calls.

insure++.leak_combine [none|**trace**|location]

Specifies how to combine leaks for the memory leak summary report. Combining by `trace` means all blocks allocated with identical stack traces will be combined into a single entry. Combining by `location` means all allocations from the same file and line (independent of the rest of the stack trace) will be combined. `none` means each allocation will be listed separately.

`insure++.leak_search [`**`on`**`|off|full]`

Specifies additional leak checking at runtime before a leak is reported. Requires that the symbol table reader be turned on. If "full" is selected, any pointer to anywhere in a block is sufficient to consider it not leaked. Selecting the “Full” option is useful when getting false leaks from 3rd party libraries.

`insure++.leak_sort [none|frequency|location|`**`size`**`]`

Specifies by what criterion the memory leak summary report is sorted. Setting this to `none` may provide better performance at exit if you have many leaks.

`insure++.leak_summary_filter`

Controls which blocks are reported in the `leaks detected at exit` and `outstanding` sections of the leak summary. For example,

```
insure++.leak_summary_filter -! * main* *
```

restricts the leak summary to those blocks with stack traces ending in `main`.

`insure++.leak_sweep [`**`on`**`|off|full]`

Specifies additional leak checking at the termination of the program. Requires that the symbol table reader be turned on. If "full" is selected, any pointer to anywhere in a block is sufficient to consider it not leaked. Selecting “Full” is useful when getting false leaks from 3rd party libraries. Leaks detected will be reported using the `summarize leaks` option (see page 163).

`insure++.leak_trace [`**`on`**`|off]`

This option determines whether or not full stack traces will be shown in the memory leak summary report.

`insure++.library_debug_ignore [library list]`

This option tells Insure++ not to read any debug (i.e. file and line) information from any library that matches any pattern in the list. The list should be a comma-separated list of shell-style glob expressions, e.g. libX*, libACE* suppresses reading from libX11.so, libXt.so, libXm.so and libACE.debug.so. This causes the application to start up more quickly.

insure++.malloc_pattern pattern

Specifies a pattern that will be written on top of memory whenever it is allocated. This pattern will be repeated for each byte in the allocated region (this option is available only if malloc_replace was selected at compile time). The default is 0, which means that no pattern will be written.

insure++.malloc_trace [-1|0|1|2|3|...|**10**|...]

Specifies the maximum number of levels to track the stack whenever a block of dynamic memory is allocated. Setting this option to a non-zero value tells Insure++ to include a description of the function call stack to at most the given depth whenever an error associated with the block is reported. Setting the value to 0 suppresses allocation stack tracing, while the value -1 traces the stack back to the main routine.

insure++.malloc_trace_threshold

This option controls how many times the parser will try to instantiate a template before giving up. It is possible to have infinite loops in a template. The default number of tries is 128; the minimum is 32.

insure++.new_overhead [0|2|4|6|8|...]

Specifies the number of bytes allocated as overhead each time new[] is called. The default is compiler-dependent, but is typically 0, 4, or 8.

insure++.pointer_slack [0|**1**|2]

This controls a heuristic in Insure++. When a pointer does not point to a valid block, but does point to an area 1 byte past the end of a valid block, does the pointer really point to that block? The value of this argument controls Insure++'s answer. The default should be changed only if Insure++ is not working correctly on your program.

Value	Meaning
0	Never assume the pointer points to the previous block
1	Assume the pointer points to the previous block if that block was dynamically allocated
2	Always assume the pointer points to the previous block. This tends to be incorrect for stack and global variables, since they are usually adjacent in memory

insure++.realloc_stress [**on**|off]

If enabled, all calls to `realloc` will cause the block in question to move. This can be useful in triggering certain kinds of bugs where the possibility of `realloc` changing addresses was not considered.

insure++.report_banner [**on**|off]

Controls whether or not a message is displayed on your terminal, reminding you that error messages have been redirected to a file. (See page 60) insure++.report_file [filename|Insra|**stderr**]

Specifies the name of the report file. Environment variables and various pattern generation keys may appear in `filename`. (See page 135) Use of the special filename Insra tells Insure++ to send its output to Insra.

insure++.report_limit [-1|0|**1**|2|3|...]

Displays only the first given number of errors of each type at any particular source line. Setting this option to `-1` will show all errors. Setting it to `0` will only show errors in summary reports, and not at runtime. (See page 64)

insure++.report_overwrite [**on**|off]

If set to `off`, error messages are appended to the report file rather than overwriting it on each run.

insure++.runtime [**on**|off]

If set to `off`, no runtime checking or profiling is performed. The program then runs much faster. This can be used to check to see if a particular fix has cured a problem, without recompiling the application without Insure++.

insure++.signal_catch [all|sig1 sig2 …]

Specifies a list of UNIX signals which Insure++ will trap. When these signals are detected, Insure++ displays a function call stack. Signals may be specified by number or by their symbolic names, with or without the `SIG` prefix. See page 41 for more details.

insure++.signal_ignore [all|sig1 sig2 …]

Specifies a list of UNIX signals which Insure++ will ignore. Insure++ makes no attempt to trap these signals. Signals may be specified by number or by their symbolic names, with or without the `SIG` prefix. (See page 41)

You will want to use this if your program is expecting any of the signals that Insure++ catches by default.

`insure++.source_path dir1 dir2 dir3`

This option takes a list of directories in which to search for source files. (See page 66) This will only be necessary if your source code has moved since it was compiled, as Insure++ remembers where all your source files are located.

`insure++.stack_limit [`**`-1`**`|0|1|2|3|...]`

Truncates runtime stack traces after displaying at most the given number of levels. Setting the option to `-1` displays all levels. Setting the option to `0` disables stack tracing. (See page 65)

`insure++.summarize [bugs] [coverage] [leaks] [outstanding]`

Generates a summary report of errors (see page 73), memory leaks (see page 75), outstanding allocated memory blocks, or coverage analysis (see page 77). In the latter case, the `coverage_switches` option (see the TCA manual) is consulted to decide how to present coverage data. The `leaks` and `outstanding` reports are affected by the `leak_combine`, `leak_sort`, and `leak_trace` options. With no arguments, this option will summarize the `bugs` and `leaks` summaries. This option has changed slightly in versions 3.1 and higher. The old leak defaults are equivalent to `leak_combine location`, `leak_sort location`, `leak_trace off`. The old `detailed` option is replaced by `leak_trace on`.

`insure++.summarize_on_error [`**`0`**`|1|2|3|...]`

Specifies how many errors must be generated before a summary (if requested) is printed. The default is `0`, which means that summaries are always printed on demand. If the number is `1` or higher, summaries are only printed if *at least* the given number of bugs (or leaks) occurred. Suppressed errors do not count towards this number. If no argument is given with this option, a value of `1` is assumed.

`insure++.suppress code [{context}]`

Suppress error messages matching the given error code and occurring in the (optionally) specified context. (See page 67, *et seq.*)

`insure++.symbol_banner [on|`**`off`**`]`

If set, Insure++ displays a message indicating that the program's symbol table is being processed whenever an application starts.

`insure++.symbol_sizes [`**`on`**`|off]`

Tells Insure++ to read variable size information from the executable and all shared libraries used by the program. This will cause the application to start up more slowly.

`insure++.symbol_sizes_ignore [library list]`

Tells Insure++ not to read variable size information from any library that matches any pattern in the list. See also library_debug_ignore. This causes the application to start up more quickly.

`insure++.symbol_strategy [on|`**`off`**`]`

Symbolstrategy API allows for faster startup.

`insure++.symbol_table [`**`on`**`|off]`

If set to `on`, Insure++ will read the executable symbol table at startup. This enables Insure++ to generate full stack traces for third party libraries as well as for code compiled with Insure++. If this option is turned off, the stack traces will show only functions compiled with Insure++, but the application will use less dynamic memory and be faster on startup. To get filenames and line numbers in stack traces with this option off, you must compile your program with the `stack_internal on` option. (See page 151)

`insure++.trace [on|`**`off`**`]`

Turns program tracing on and off. In order to get file names and line numbers in the trace output, you must have the `stack_internal on` option set when compiling the program. See the Tracing section on page 37 of this manual for more information about program tracing.

`insure++.trace_banner [`**`on`**`|off]`

Specifies whether to print message at runtime showing file to which the trace output will be written.

`insure++.trace_file [filename|`**`stderr`**`]`

Specifies filename to which the trace output will be written. filename may use the same special tokens shown on page 134.

`insure++.unsuppress code [{context}]`

Enables error messages matching the given error codes and occurring in the (optionally) specified context. (See page 72, *et seq*.)

Options used by Insra

Running insra

insra.body_background_color [**White**|color]

Specifies the color Insra will use for the message body area background.

insra.body_font [**Fixed**|font]

Specifies the font Insra will use for the message body text. On some systems, e.g. SGI, the Fixed font is much too large. This option can be used to select a smaller font.

insra.body_height [0|1|2|...|**8**|...]

Specifies the starting height of the Insra message window in number of rows of visible text. This may be modified while Insra is running using the standard Motif controls.

insra.body_text_color [**Black**|color]

Specifies the color Insra will use for the message body text.

insra.body_width [0|1|2|...|**80**|...]

Specifies the starting width of the Insra message window n number of columns of visible text. This may be modified while Insra is running using the standard Motif controls. If this option is set to a different value than header_width, the larger value will be used.

insra.header_background_color [**White**|color]

Specifies the color Insra will use for the message header area background.

insra.header_font [**Fixed**|font]

Specifies the font Insra will use for the message header text. On some systems, e.g. SGI, the Fixed font is much too large. This option can be used to select a smaller font.

insra.header_height [0|1|2|...|**8**|...]

Specifies the starting height of the Insra message header area in number of rows of visible text. This may be modified while Insra is running using the standard Motif controls.

`insra.header_highlight_color`
`[`**`LightSteelBlue2`**`|color]`

Specifies the color Insra will use to indicate the currently selected message or session header in the message header area.

`insra.header_highlight_text_color [`**`Black`**`|color]`

Specifies the color Insra will use for the text of the currently selected message or session header in the message header area.

`insra.header_session_color [`**`LightSkyBlue3`**`|color]`

Specifies the color Insra will use to indicate a session header.

`insra.header_session_text_color [`**`Black`**`|color]`

Specifies the color Insra will use for session header text.

`insra.header_text_color [`**`Black`**`|color]`

Specifies the color Insra will use for message header text.

`insra.header_width [0|1|2|...|`**`80`**`|...]`

Specifies the starting width of the Insra message header area in number of columns of visible text. This may be modified while Insra is running using the standard Motif controls. If this option is set to a different value than `body_width`, the larger value will be used.

`insra.port [`**`3255`**`|port_number]`

Specifies which port Insra should use to communicate with Insure++ and Insure++-compiled programs.

`insra.toolbar [`**`on`**`|off]`

Specifies whether Insra's toolbar is displayed. All toolbar commands can also be chosen from the menu bar.

`insra.visual [`**`xterm -e vi +%l %f`**`|emacs +%l %f|`
`other_editor_command]`

Specifies how Insra should call an editor to display the line of source code causing the error. Insra will expand the `%l` token to the line number and the `%f` token to the file name before executing the given command. It is important to include the full path of any binary that lives in a location not on your path. Setting this option with no command string disables source browsing from Insra.

Memory Overflow

One of the common errors that Insure++ detects occurs when a program reads or writes beyond the bounds of a valid memory area. This type of problem normally generates a `READ_OVERFLOW` or `WRITE_OVERFLOW` error which describes the memory regions being accessed with their addresses and sizes as shown below.

```
[hello.c:13] **WRITE_OVERFLOW**
>> strcat(str, argv[i]);

   Writing overflows memory: str

        bbbbbbbbbbbbbbbbbbbbbbbbbbbb
        |          16               |   2   |
        wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww

   Writing (w) : 0xf7fff8a8 thru 0xf7fff8b9 (18 bytes)
   To block (b): 0xf7fff8a8 thru 0xf7fff8b7 (16 bytes)
             str, declared at hello.c, 8

   Stack trace where the error occurred:
                  main() hello.c, 13

**Memory corrupted. Program may crash!!**
```

Figure 14. Sample output from a `WRITE_OVERFLOW` error

Overflow diagrams

The textual information in Figure 14 describes the memory blocks involved in the overflow operation using their memory addresses and sizes.

To gain a more intuitive understanding of the nature of the problem, a text-based “overflow diagram” is also shown. This pattern attempts to demonstrate the nature and extent of the problem by representing the memory blocks involved pictorially.

```
bbbbbbbbbbbbbbbbbbbbbbbbbbbb
|          16              |   2   |
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
```

In this case, the row of `b` characters represents the available memory block, while the row of `w`’s shows the range of memory addresses being written. The block being written is longer than the actual memory block, which causes the error.

The numbers shown indicate the size, in bytes, of the various regions and match those of the textual error message.

The relative length and alignment of the rows of characters is intended to indicate the size and relative positioning of the memory blocks which cause the error. The above case shows both blocks beginning at the same position with the written block extending beyond the end of the memory region. If the region being written extended both before and after the available block, a diagram such as the following would have been displayed.

```
       bbbbbbbbbbbbbbbbbbbbbbbbbbbb
|  5   |          16              |   2   |
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
```

Completely disjoint memory blocks are indicated by a diagram of the form

```
                             bbbbbbbbbbbbbbbbbb
|  4  |         40           |       16       |
wwwwwww
```

Similar diagrams appear for both `READ_OVERFLOW` and `WRITE_OVERFLOW` errors. In the former case, the block being read is represented by a row of `r` characters instead of `w`’s. Similarly, the memory regions involved in parameter size mismatch errors are indicated using a row of `p` characters for the parameter block. (See `PARM_BAD_RANGE`)

Error Codes

This section is intended to provide a reference for the various error messages generated by Insure++.

This appendix consists of two parts.

The first lists each error code alphabetically together with its interpretation and an indication of whether or not it is suppressed by default.

The second gives a detailed description of each error including:

- A brief explanation of what problem has been detected.
- An example program that generates a similar error.
- Output that would be generated by running the example, with annotations indicating what the various pieces of the diagnostic mean and how they should be interpreted in identifying your own problems.

  Note that the exact appearance of the error message depends on how Insure++ is configured.
- A brief description of ways in which the problem might be eliminated.

Note that sometimes you will see values identified as “`<argument #>`” or “`<return>`” instead of names from your program. In this case, `<argument n>` refers to the `n`th argument passed to the current function (i.e. the one where the error was detected), and `<return>` refers to a value returned from the function indicated.

C++

Code	Meaning	Enabled?
`ALLOC_CONFLICT`	Mixing malloc/free with new/delete	X
`(badfree)`	Free called on block allocated with new	X
`(baddelete)`	Delete called on block allocated with malloc	X
`BAD_CAST`	Cast of pointer loses precision	X
`BAD_DECL`	Incompatible global declarations	X
`BAD_FORMAT`	Mismatch in format specification	
`(sign)`	`int` vs. `unsigned int`	
`(compatible)`	`int` vs. `long`, both same size	
`(incompatible)`	`int` vs. `double`	X
`(other)`	Wrong number of arguments	X
`BAD_INTERFACE`	Declaration of function in interface conflicts with declaration in program	X
`BAD_PARM`	Mismatch in argument type	
`(sign)`	`int` vs. `unsigned int`	
`(compatible)`	`int` vs. `long`, both same size	
`(incompatible)`	`int` vs. `double`	X
`(pointer)`	All pointers are equivalent	X

Code	Meaning	Enabled?
(union)	Require exact match on unions	X
(other)	Wrong number of arguments	X
COPY_BAD_RANGE	Attempt to copy out-of-range pointer	X
COPY_DANGLING	Attempt to copy dangling pointer	X
COPY_UNINIT_PTR	Attempt to copy uninitialized pointer	X
COPY_WILD	Attempt to copy wild pointer	X
DEAD_CODE		
(emptyloopbody)	Loop body is empty	X
(emptystmt)	Statement is empty	X
(noeffect)	Code has no effect	X
(notevaluated)	Code is not evaluated	X
DELETE_MISMATCH	Mismatch between new/new[] and delete/delete[]	X
(bracket)	new, delete[]	X
(nobracket)	new[], delete	X
EXPR_BAD_RANGE	Expression exceeded range	X

Code	Meaning	Enabled?
EXPR_DANGLING	Expression uses dangling pointer	X
EXPR_NULL	Expression uses NULL pointer	X
EXPR_UNINIT_PTR	Expression uses uninitialized pointer	X
EXPR_UNRELATED_PTRCMP	Expression compares unrelated pointers	X
EXPR_UNRELATED_PTRDIFF	Expression subtracts unrelated pointers	X
EXPR_WILD	Expression uses wild pointer	
FREE_BODY	Freeing memory block from body	X
FREE_DANGLING	Freeing dangling pointer	X
FREE_GLOBAL	Freeing global memory	X
FREE_LOCAL	Freeing local memory	X
FREE_NULL	Freeing NULL pointer	X
FREE_UNINIT_PTR	Freeing uninitialized pointer	X
FREE_WILD	Freeing wild pointer	X
FUNC_BAD	Function pointer is not a function	X
FUNC_NULL	Function pointer is NULL	X

Code	Meaning	Enabled?
FUNC_UNINIT_PTR	Function pointer is uninitialized	X
FUNC_WILD	Function pointer is wild	X
INSURE_ERROR	Internal error	X
INSURE_WARNING	Output from iic_warning	
LEAK_ASSIGN	Memory leaked due to pointer reassignment	X
LEAK_FREE	Memory leaked freeing block	X
LEAK_RETURN	Memory leaked by ignoring return value	X
LEAK_SCOPE	Memory leaked leaving scope	X
PARM_BAD_RANGE	Array parameter exceeded range	X
PARM_DANGLING	Array parameter is dangling pointer	X
PARM_NULL	Array parameter is NULL	X
PARM_UNINIT_PTR	Array parameter is uninitialized pointer	X
PARM_WILD	Array parameter is wild	X
READ_BAD_INDEX	Reading array out of range	X
READ_DANGLING	Reading from a dangling pointer	X

Code	Meaning	Enabled?
`READ_NULL`	Reading `NULL` pointer	X
`READ_OVERFLOW`		
`(normal)`	Reading overflows memory	X
`(nonull)`	String is not NULL-terminated within range	X
`(string)`	Alleged string does not begin within legal range	X
`(struct)`	Structure reference out of range	X
`(maybe)`	Dereferencing structure of improper size (may be o.k.)	
`READ_UNINIT_MEM`	Reading uninitialized memory	
`(copy)`	Copy from uninitialized region	
`(read)`	Use of uninitialized value	X
`READ_UNINIT_PTR`	Reading from uninitialized pointer	X
`READ_WILD`	Reading wild pointer	X
`RETURN_DANGLING`	Returning pointer to local variable	X
`RETURN_FAILURE`	Function call returned an error	
`RETURN_INCONSISTENT`	Function returns inconsistent value	
`(level 1)`	No declaration, returns nothing	

Code	Meaning	Enabled?
(level 2)	Declared `int` returns nothing	X
(level 3)	Declared non-`int`, returns nothing	X
(level 4)	Returns different types at different statements	X
UNUSED_VAR	Unused variables	
(assigned)	Assigned but never used	
(unused)	Never used	
USER_ERROR	User generated error message	X
VIRTUAL_BAD	Error in runtime initialization of virtual functions	X
WRITE_BAD_INDEX	Writing array out of range	X
WRITE_DANGLING	Writing to a dangling pointer	X
WRITE_NULL	Writing to a `NULL` pointer	X
WRITE_OVERFLOW		
(normal)	Writing overflows memory	X
(struct)	Structure reference out of range	X
(maybe)	Dereferencing structure of improper size (may be o.k.)	

Code	Meaning	Enabled?
`WRITE_UNINIT_PTR`	Writing to an uninitialized pointer	X
`WRITE_WILD`	Writing to a wild pointer	X

ALLOC_CONFLICT

Memory allocation conflict

C++

This error is generated when a memory block is allocated with `new` (`malloc`) and freed with `free` (`delete`).

Insure++ distinguishes between the two possibilities as follows:

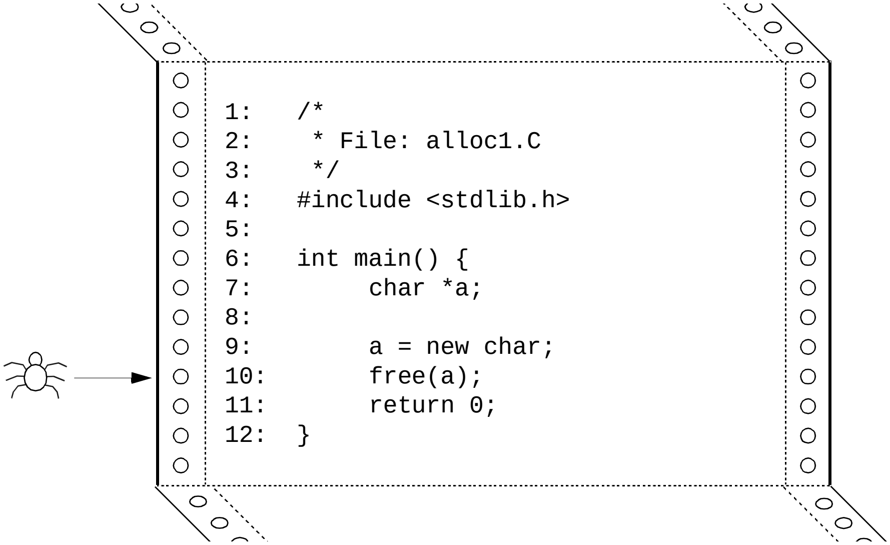
`badfree` Memory was allocated with `new` or `new[]` and an attempt was made to free it with `free`.

`baddelete` Memory was allocated with `malloc` and an attempt was made to free it with `delete` or `delete[]`.

Some compilers do allow this, but it is not good programming practice and could be a portability problem.

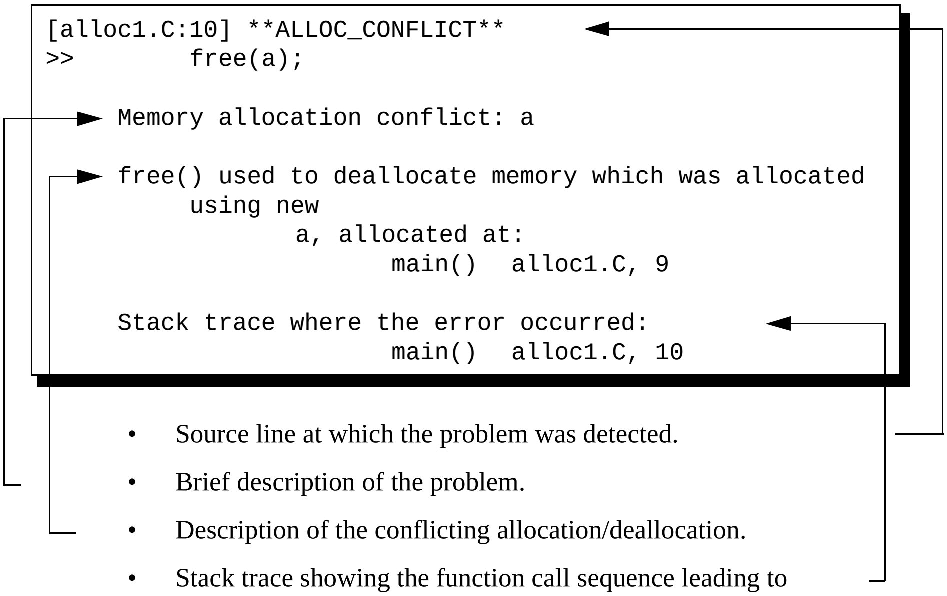
Problem #1

The following code shows a typical example of allocating a block of memory with `new` and then freeing it with `free`, instead of `delete`.



```
/*
 * File: alloc1.C
 */
#include <stdlib.h>

int main() {
     char *a;

     a = new char;
     free(a);
     return 0;
}
```

Diagnosis (at runtime)

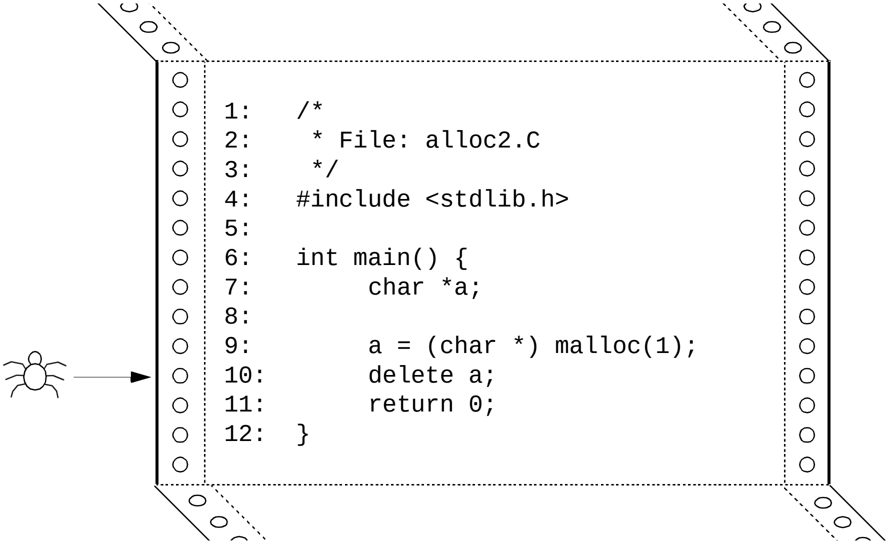


```
[alloc1.C:10] **ALLOC_CONFLICT**
>>        free(a);

     Memory allocation conflict: a

     free() used to deallocate memory which was allocated
          using new
                 a, allocated at:
                       main()  alloc1.C, 9

     Stack trace where the error occurred:
                       main()  alloc1.C, 10
```

- Source line at which the problem was detected.
- Brief description of the problem.
- Description of the conflicting allocation/deallocation.
- Stack trace showing the function call sequence leading to the error.

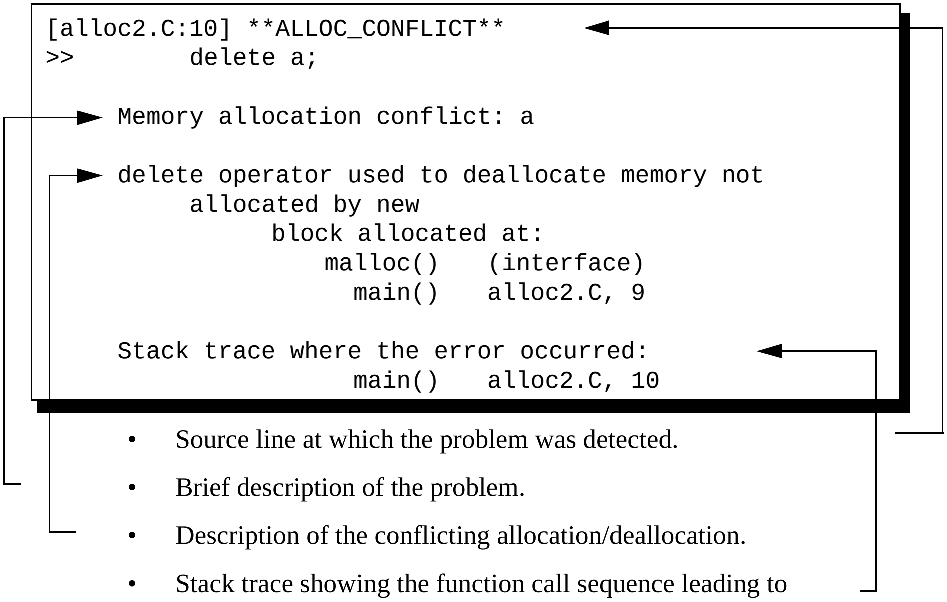
Problem #2

The following code shows another typical example of this type of error, allocating a block of memory with `malloc` and then freeing it with `delete`.



```
/*
 * File: alloc2.C
 */
#include <stdlib.h>

int main() {
     char *a;

     a = (char *) malloc(1);
     delete a;
     return 0;
}
```

Diagnosis (at runtime)



```
[alloc2.C:10] **ALLOC_CONFLICT**
>>        delete a;

     Memory allocation conflict: a

     delete operator used to deallocate memory not
          allocated by new
                block allocated at:
                   malloc()   (interface)
                     main()   alloc2.C, 9

     Stack trace where the error occurred:
                     main()   alloc2.C, 10
```

- Source line at which the problem was detected.
- Brief description of the problem.
- Description of the conflicting allocation/deallocation.
- Stack trace showing the function call sequence leading to the error.

Repair

This type of error can be corrected by making sure that all your memory allocations match up.

BAD_CAST

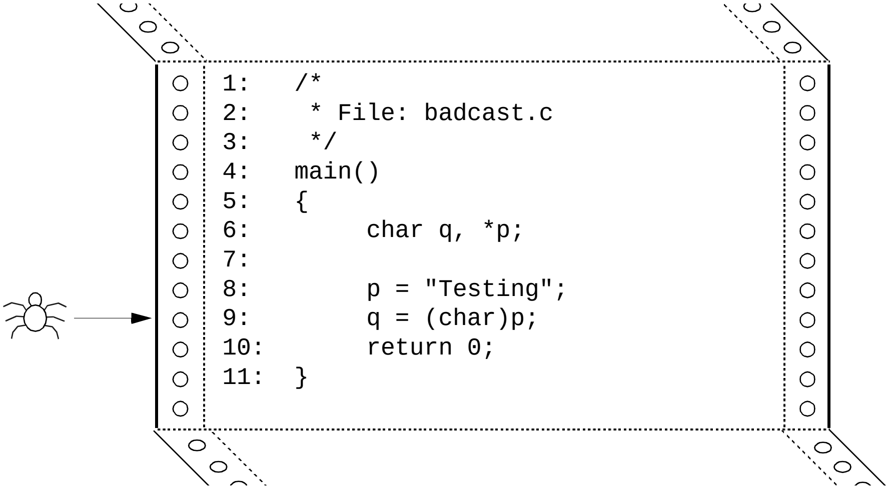
Cast of pointer loses precision

Porting code between differing machine architectures can be difficult for many reasons. A particularly tricky problem occurs when the sizes of data objects, particularly pointers, differ from that for which the software was created. This error occurs when a pointer is cast to a type with fewer bits, causing information to be lost, and is designed to help in porting codes to architectures where, for example, pointers and integers are of different lengths.

Note that compilers will often catch this problem unless the user has “carefully” added the appropriate typecast to make the conversion “safe”.

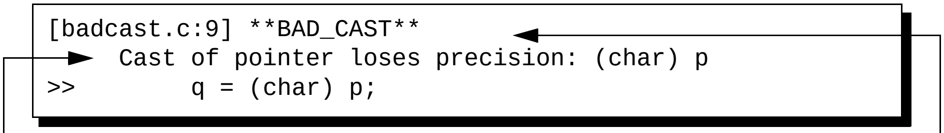
Problem

The following code shows a pointer being copied to a variable too small to hold all its bits.



```
/*
 * File: badcast.c
 */
main()
{
     char q, *p;

     p = "Testing";
     q = (char)p;
     return 0;
}
```

Diagnosis (during compilation)



```
[badcast.c:9] **BAD_CAST**
    Cast of pointer loses precision: (char) p
>>        q = (char) p;
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.

Repair

This error normally indicates a significant portability problem that should be corrected by using a different type to save the pointer expression. In ANSI C the type `void *` will always be large enough to hold a pointer value.

BAD_DECL

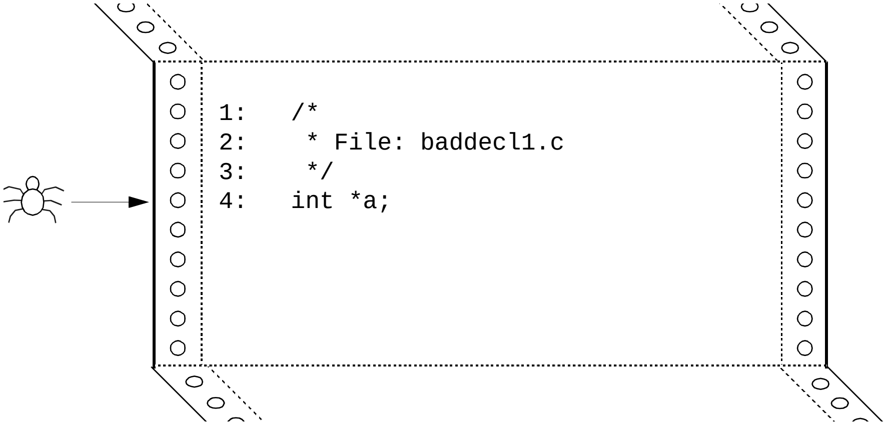
Global declarations are inconsistent

This error is generated whenever Insure++ detects that a variable has been declared as two different types in distinct source files. This can happen when there are two conflicting definitions of an object or when an extern reference to an object uses a different type than its definition.

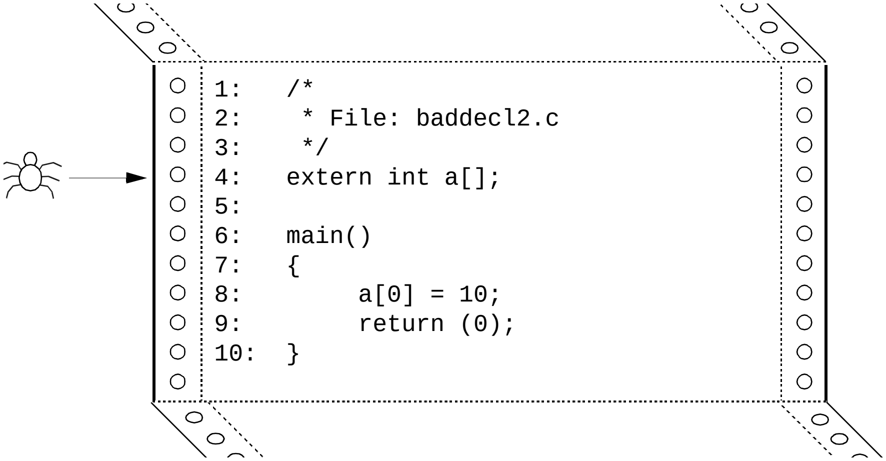
In any case, Insure++ proceeds as though the variable *definition* is correct, overriding the extern reference.

Problem

In the following example, the file baddecl1.c declares the variable a to be a pointer, while the file baddecl2.c declares it to be an array type.

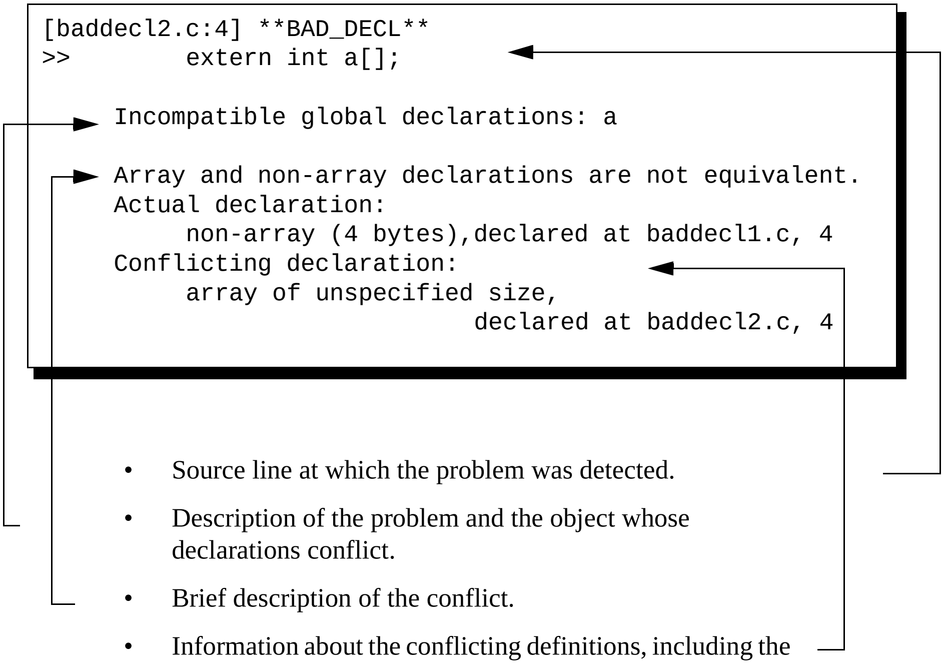


```
/*
 * File: baddecl1.c
 */
int *a;
```



```
/*
 * File: baddecl2.c
 */
extern int a[];

main()
{
     a[0] = 10;
     return (0);
}
```

Diagnosis (at runtime)



```
[baddecl2.c:4] **BAD_DECL**
>>        extern int a[];

      Incompatible global declarations: a

      Array and non-array declarations are not equivalent.
      Actual declaration:
           non-array (4 bytes),declared at baddecl1.c, 4
      Conflicting declaration:
           array of unspecified size,
                               declared at baddecl2.c, 4
```

- Source line at which the problem was detected.
- Description of the problem and the object whose declarations conflict.
- Brief description of the conflict.
- Information about the conflicting definitions, including the sizes of the declared objects and the locations of their declarations.

Repair

The lines on which the conflicting declarations are made are both shown in the diagnostic report. They should be examined and the conflict resolved.

In the case shown here, for example, a suitable correction would be to change the declaration file to declare an array with a fixed size, e.g.,

```
baddecl1.c, 4: int a[10];
```

An alternative correction would be to change the definition in baddecl2.c to indicate a pointer variable, e.g.,

```
baddecl2.c, 4: extern int *a;
```

Note that this change on its own will not fix the problem. In fact, if you ran the program modified this way, you would get another error, EXPR_NULL, because the pointer, a, doesn't actually point to anything and is NULL by virtue of being a global variable, initialized to zero.

To make this version of the code correct, you would need to include something to allocate memory and store the pointer in a, e.g.,

```
/*
 * File: baddecl2.c (modified)
 */
#include <stdlib.h>
extern int *a;

main()
{
    a = (char *)malloc(10*sizeof(int));
    a[0] = 10;
}
```

Some applications may genuinely need to declare objects with different sizes, in which case you can suppress error messages by inserting the line

```
insure++.suppress BAD_DECL
```

into your .psrc file.

BAD_FORMAT

Mismatch in format specification

This error is generated when a call to one of the `printf` or `scanf` routines contains a mismatch between a parameter type and the corresponding format specifier or the format string is nonsensical.

Insure++ distinguishes several types of mismatch which have different levels of severity as follows:

`sign` Types differ only by sign, e.g., `int` vs. `unsigned int`.

`compatible` Fundamental types are different but they happen to have the same representation on the particular hardware in use, e.g., `int` vs. `long` on machines where both are 32-bits, or `int *` vs. `long` where both are 32-bits.

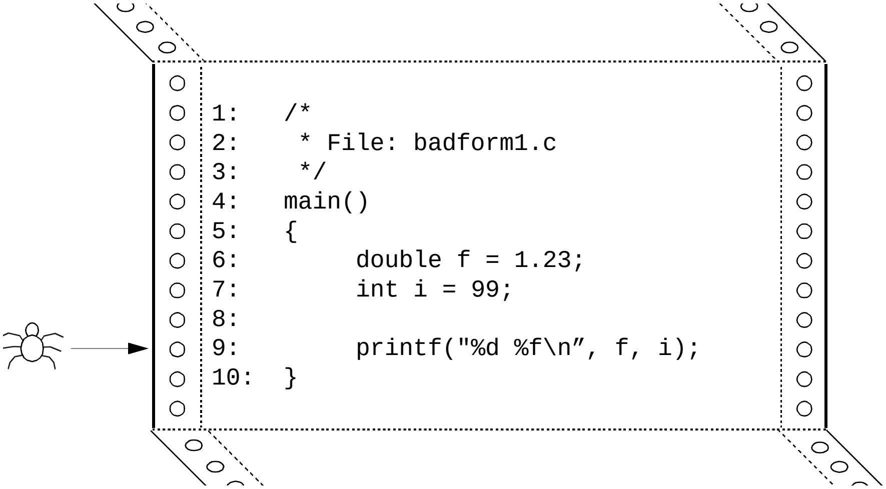
`incompatible` Fundamental types are different, e.g. `int` vs. `double`.

`other` A problem other than an argument type mismatch is detected, such as passing the wrong number of arguments.

Error messages are classified according to this scheme and can be selectively enabled or disabled as described in the “Repair” section on page 196.

Problem #1

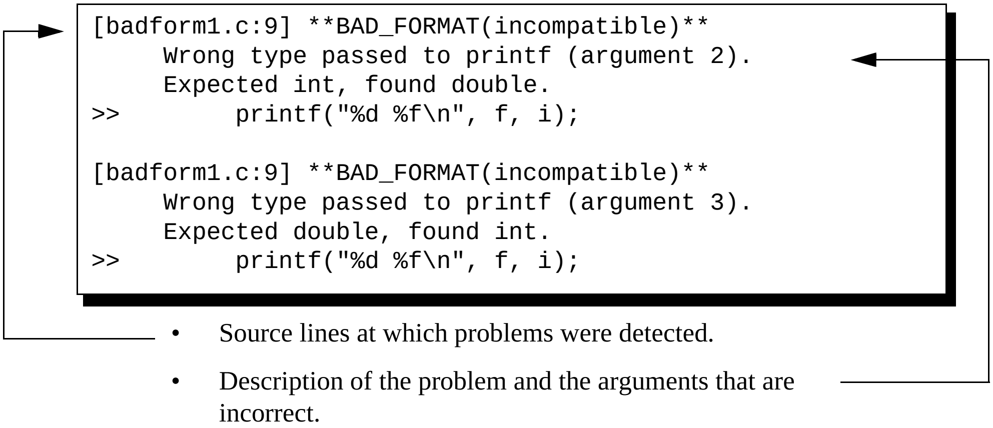
An example of format type mismatch occurs when the format specifiers passed to one of the `printf` routines do not correspond to the data, as shown below.



```
/*
 * File: badform1.c
 */
main()
{
     double f = 1.23;
     int i = 99;

     printf("%d %f\n”, f, i);
}
```

This type of mismatch is detected during compilation.

Diagnosis (during compilation)

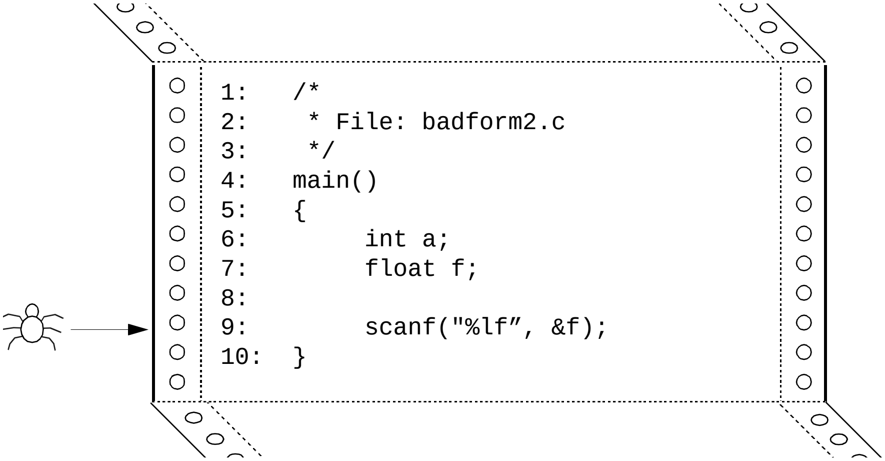


```
[badform1.c:9] **BAD_FORMAT(incompatible)**
     Wrong type passed to printf (argument 2).
     Expected int, found double.
>>        printf("%d %f\n", f, i);

[badform1.c:9] **BAD_FORMAT(incompatible)**
     Wrong type passed to printf (argument 3).
     Expected double, found int.
>>        printf("%d %f\n", f, i);
```

- Source lines at which problems were detected.
- Description of the problem and the arguments that are incorrect.

Problem #2

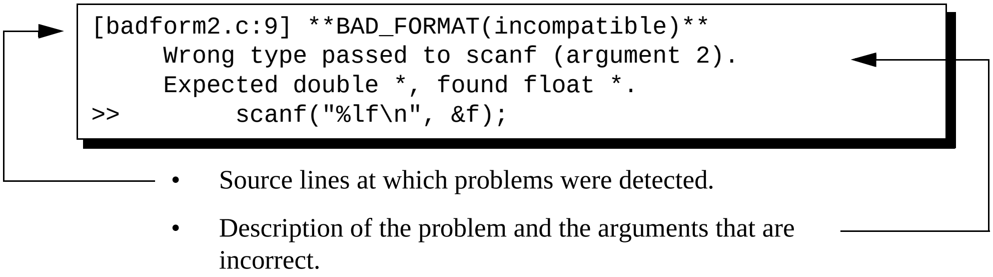
A more dangerous problem occurs when the types passed as arguments to one of the `scanf` functions are incorrect. In the following code, for example, the call to scanf tries to read a double precision value, indicated by the “%lf” format, into a single precision value. This will overwrite memory.



```
1:   /*
2:    * File: badform2.c
3:    */
4:   main()
5:   {
6:        int a;
7:        float f;
8:
9:        scanf("%lf”, &f);
10:  }
```

This problem is again diagnosed at compile time (along with the WRITE_OVERFLOW, which is not shown below).

Diagnosis (during compilation)

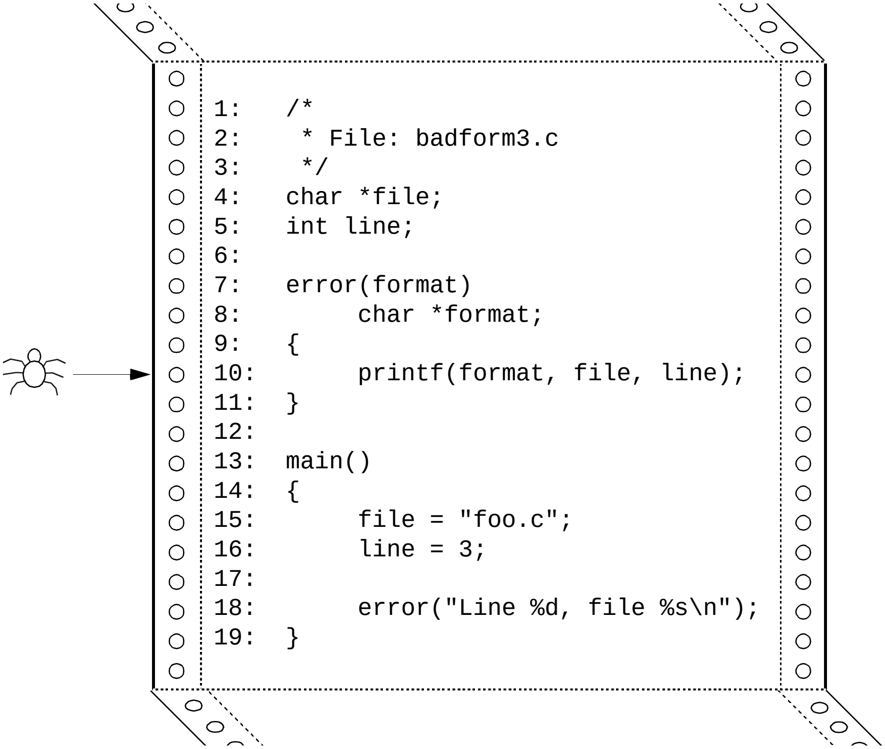


```
[badform2.c:9] **BAD_FORMAT(incompatible)**
     Wrong type passed to scanf (argument 2).
     Expected double *, found float *.
>>        scanf("%lf\n", &f);
```

- Source lines at which problems were detected.
- Description of the problem and the arguments that are incorrect.

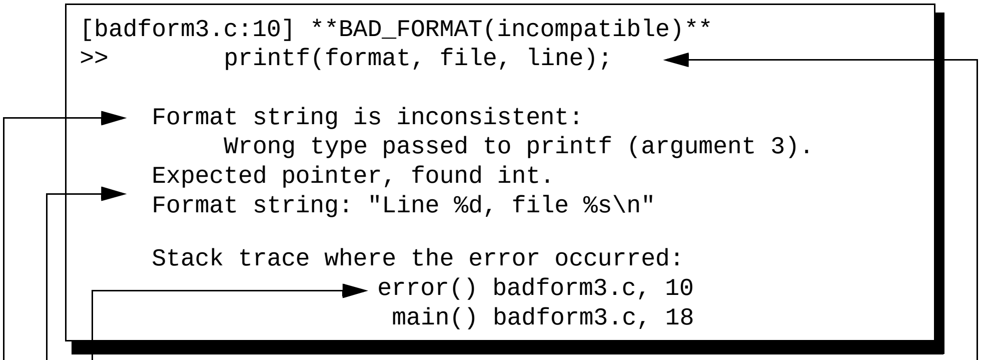
Problem #3

A third type of problem is caused when the format string being used is a variable rather than an explicit string. The following code contains an error handler that attempts to print out a message containing a filename and line number. In line 18 of the calling routine, however, the arguments are reversed.



```
1:   /*
2:    * File: badform3.c
3:    */
4:   char *file;
5:   int line;
6:
7:   error(format)
8:        char *format;
9:   {
10:       printf(format, file, line);
11:  }
12:
13:  main()
14:  {
15:       file = "foo.c";
16:       line = 3;
17:
18:       error("Line %d, file %s\n");
19:  }
```

Diagnosis (at runtime)



```
[badform3.c:10] **BAD_FORMAT(incompatible)**
>>        printf(format, file, line);

     Format string is inconsistent:
          Wrong type passed to printf (argument 3).
     Expected pointer, found int.
     Format string: "Line %d, file %s\n"

     Stack trace where the error occurred:
                     error() badform3.c, 10
                      main() badform3.c, 18
```

- Source line at which the problem was detected.
- Description of the problem and the argument that is in error.
- Explanation of the error and the format string that caused it.
- Stack trace showing the function call sequence leading to the error.

The error diagnosed in this message is in the “`incompatible`” category, because any attempt to print a string by passing an integer variable will result in garbage. Note that with some compilers, this program may cause a core dump because of this error, while others will merely produce incorrect output.

There is, however, a second potential error in this code in the same line.

Because the arguments are in the wrong order in line 7, an attempt will be made to print a pointer variable as an integer. This error is in the “`compatible`” class, since a pointer and an integer are both the same size in memory. Since “`compatible`” `BAD_FORMAT` errors are suppressed by default, you will not see it. (These errors are suppressed because they will cause unexpected rather than incorrect behavior.)

If you enabled these errors, you would see a second problem report from this code.

If you run Insure++ on an architecture where pointers and integers are not the same length, then this second error would also be in the “`incompatible`” class and would be displayed by default.

Repair

Most of these problems are simple to correct based on the information given. Normally, the correction is one or more of the following

- Change the format specifier used in the format string.
- Change the type of the variable involved.
- Add a suitable typecast.

For example, problem #1 can be corrected by simply changing the incorrect line of code as follows

```
badform1.c, line 9:printf("%d %f\n", i, f);
```

The other problems can be similarly corrected.

If your application generates error messages that you wish to ignore, you can add the option

```
insure++.suppress BAD_FORMAT
```

to your `.psrc` file.

This directive suppresses all `BAD_FORMAT` messages. If you wish to be more selective and suppress only a certain type of error, you can use the syntax

```
insure++.suppress BAD_FORMAT(class1, class2, …)
```

where the arguments are one or more of the identifiers for the various categories of error described on page 190.

Similarly, you can enable suppressed types with an unsuppress command. The problem with the pointer and integer that was not shown in the current example could be displayed by adding the option

```
insure++.unsuppress BAD_FORMAT(compatible)
```

to your .psrc file. For an example of this option, as well as the remaining subcategories of BAD_FORMAT, see the example badform4.c.

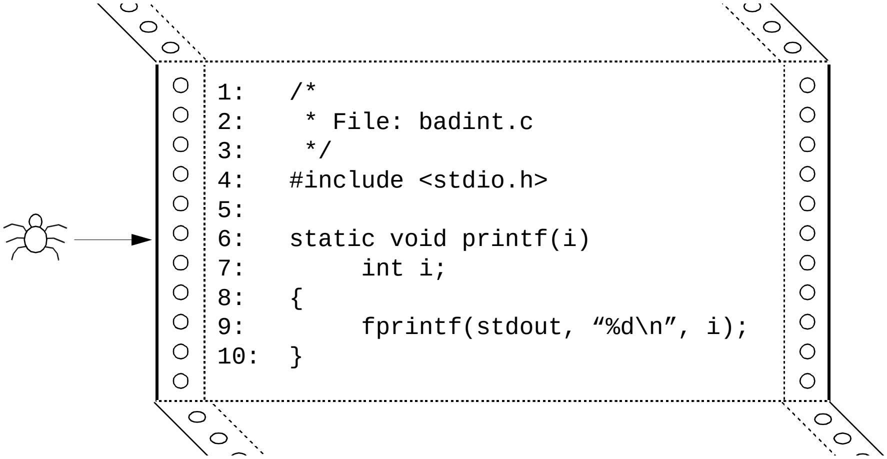
BAD_INTERFACE

Actual declaration of xxx conflicts with interface, or Ignoring interface for xxx: conflicts with static or in-line declaration

This error will be generated any time there is a significant discrepancy between the source code being processed and an interface to one of the functions in the code. Common sources of this problem are redeclarations of standard system functions in your code.

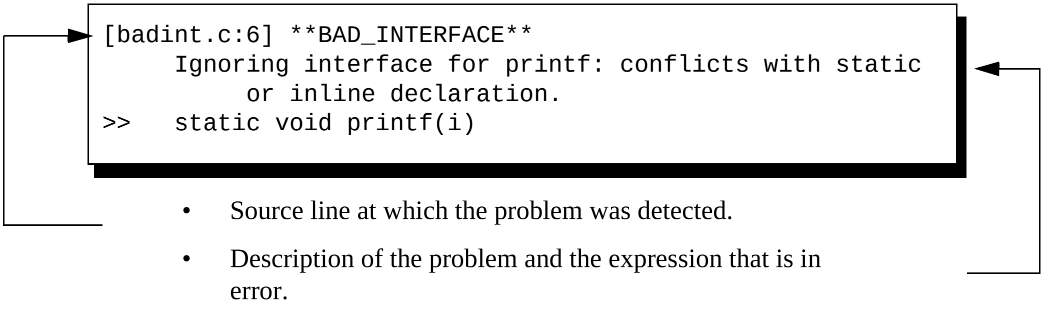
Problem

The following code shows a redeclaration of the function `printf` which will conflict with the version of the function expected by the interface.



```
/*
 * File: badint.c
 */
#include <stdio.h>

static void printf(i)
     int i;
{
     fprintf(stdout, “%d\n”, i);
}
```

Diagnosis (during compilation)



```
[badint.c:6] **BAD_INTERFACE**
     Ignoring interface for printf: conflicts with static
          or inline declaration.
>>   static void printf(i)
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.

Repair

There are several ways to approach solving this problem. The correct solution for your situation depends upon why the function was redefined in your code. If this is a version of the function that is used with all of your code, a permanent solution would be to write a new interface corresponding to your version of the function. (See “Writing simple interfaces” on page 112.) A quicker, more temporary solution, appropriate if you only use this version of the function occasionally, would be to temporarily disable the checking of this interface using the `interface_ignore` option in your `.psrc` file. This option can be turned on and off on a per file basis as you work with different code which uses different versions of the function in question.

BAD_PARM

Mismatch in argument type

This error is generated when an argument to a function or subroutine does not match the type specified in an earlier declaration or an interface file.

Insure++ distinguishes several types of mismatch which have different levels of severity as follows:

`sign` Types differ only by sign, e.g., `int` vs. `unsigned int`.

`compatible` Fundamental types are different but they happen to have the same representation on the particular hardware in use, e.g., `int` vs. `long` on machines where both are 32-bits.

`incompatible` Fundamental types are different, e.g. `int` vs. `float`.

`union` Forces a declared union argument to match only a similar union as an actual argument. If this is suppressed, you may pass any of the individual union elements to the routine, rather than the union type, or pass a union to a routine which expects one of the union-elements as an argument.

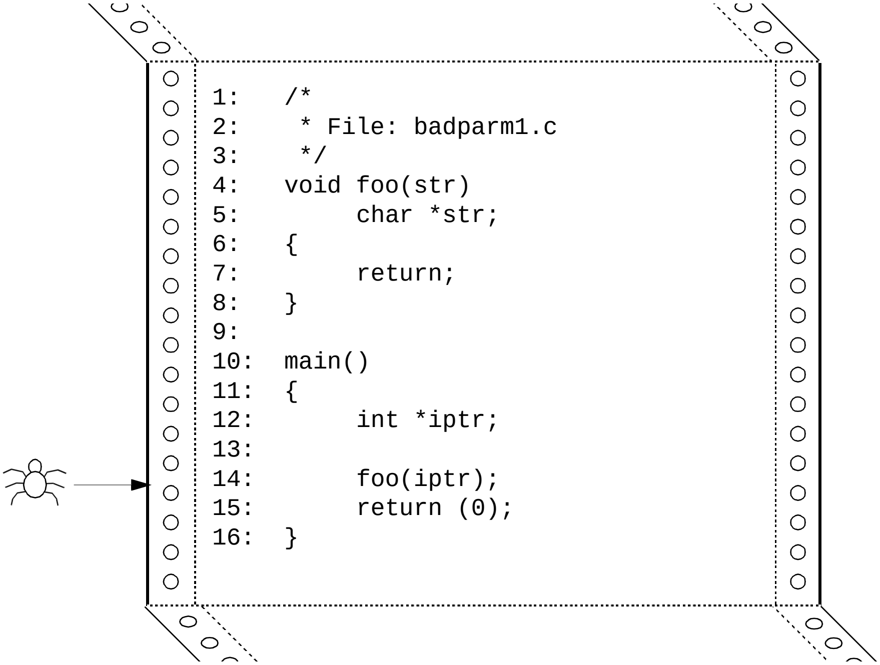
`other` An error was detected that is not simply a mismatched argument type, such as passing the wrong number of arguments to a function.

`pointer` This is not an error class, but a keyword used to suppress messages about mismatched pointer types, such as `int *` vs. `char *`. See page 207.

Error messages are classified according to this scheme and can be selectively enabled or disabled as described in the “Repair” section on page 207.

Problem #1

The following shows an error in which an incorrect argument is passed to the function `foo`.



```
/*
 * File: badparm1.c
 */
void foo(str)
     char *str;
{
     return;
}

main()
{
     int *iptr;

     foo(iptr);
     return (0);
}
```

This type of mismatch is detected during compilation.

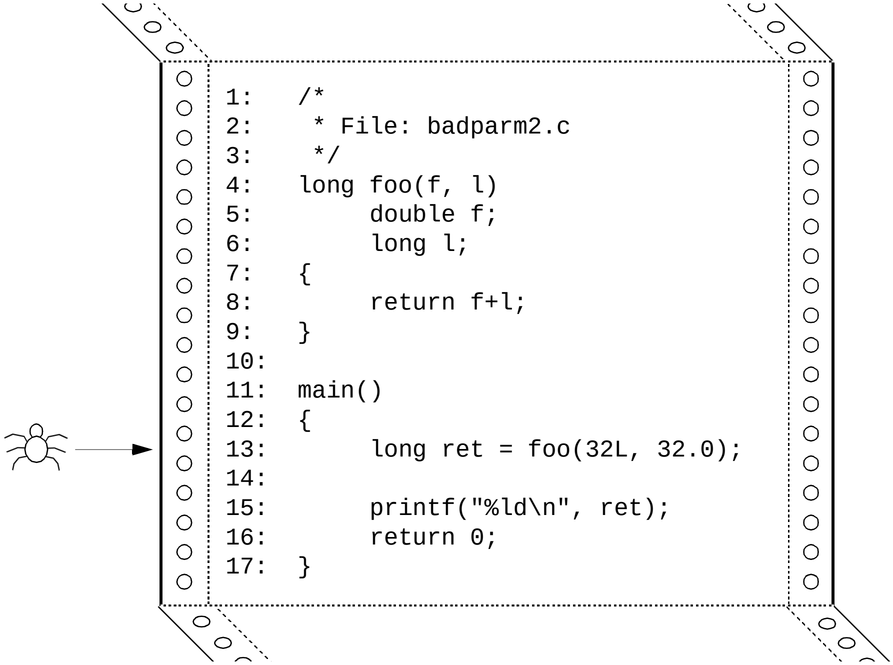
Diagnosis (during compilation)

```
[badparm1.c:14] **BAD_PARM(incompatible)**
     Wrong type passed to foo (argument 1: str)
     Expected char *, found int *.
>>        foo(iptr)
```

- Source lines at which problems were detected.
- Description of the problem and the arguments that are incorrect.

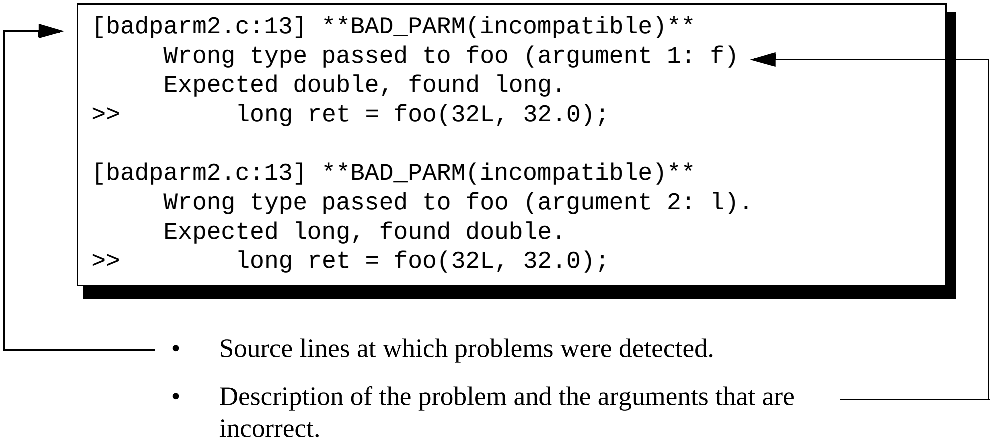
Problem #2

Another simple problem occurs when arguments are passed to functions in the wrong order, as in the following example.



```
/*
 * File: badparm2.c
 */
long foo(f, l)
     double f;
     long l;
{
     return f+l;
}

main()
{
     long ret = foo(32L, 32.0);

     printf("%ld\n", ret);
     return 0;
}
```

Diagnosis (during compilation)



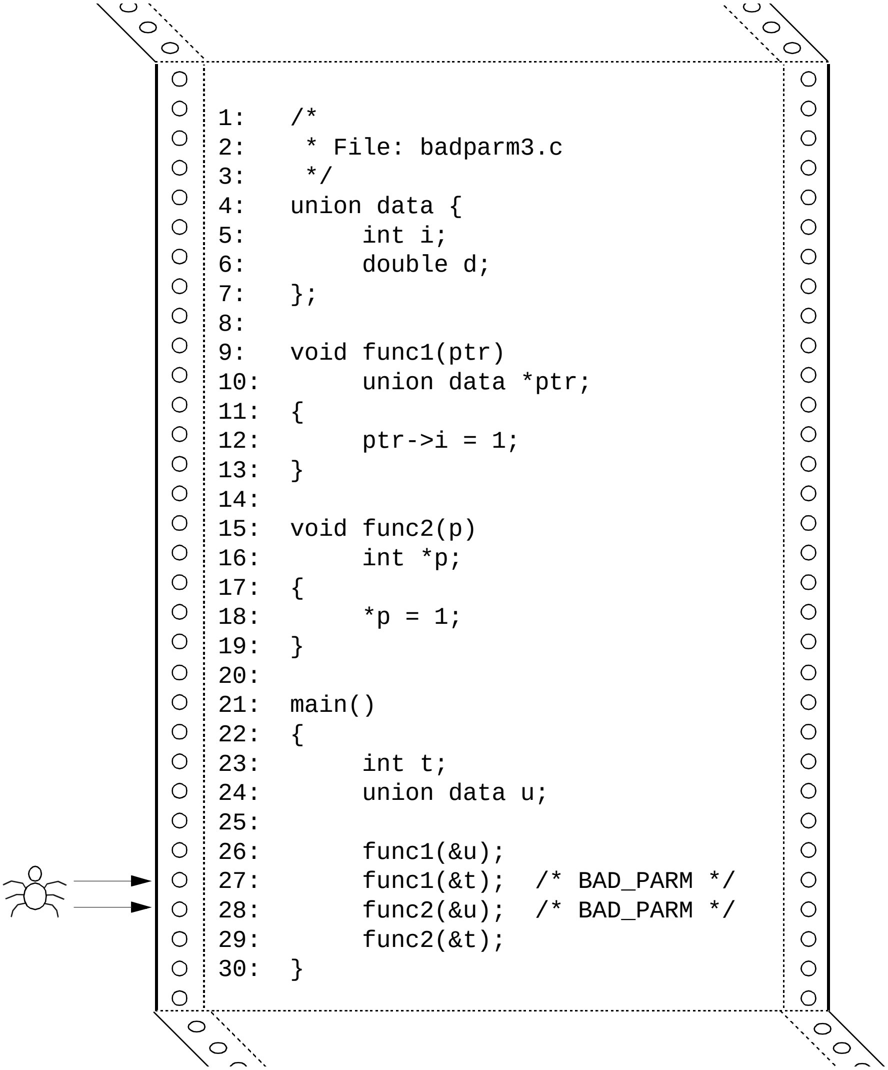
```
[badparm2.c:13] **BAD_PARM(incompatible)**
     Wrong type passed to foo (argument 1: f)
     Expected double, found long.
>>        long ret = foo(32L, 32.0);

[badparm2.c:13] **BAD_PARM(incompatible)**
     Wrong type passed to foo (argument 2: l).
     Expected long, found double.
>>        long ret = foo(32L, 32.0);
```

- Source lines at which problems were detected.
- Description of the problem and the arguments that are incorrect.

Problem #3

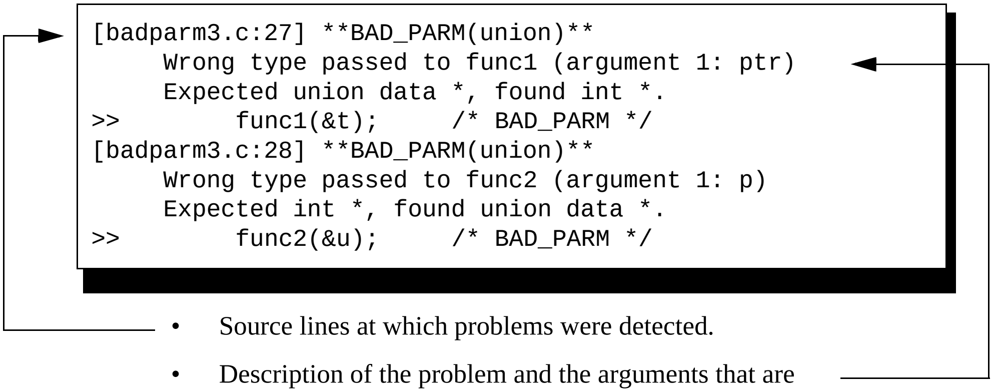
The following example illustrates the BAD_PARM(union) error category. The functions func1 and func2 expect to be passed a union and a pointer to an integer, respectively. The code in the main routine then invokes the two functions both properly and by passing the incorrect types.

Note that this code will probably work on most systems due to the internal alignment of the various data types. Relying on this behavior is, however, non-portable.



```
/*
 * File: badparm3.c
 */
union data {
        int i;
        double d;
};

void func1(ptr)
        union data *ptr;
{
        ptr->i = 1;
}

void func2(p)
        int *p;
{
        *p = 1;
}

main()
{
        int t;
        union data u;

        func1(&u);
        func1(&t);  /* BAD_PARM */
        func2(&u);  /* BAD_PARM */
        func2(&t);
}
```

Diagnosis (during compilation)



```
[badparm3.c:27] **BAD_PARM(union)**
     Wrong type passed to func1 (argument 1: ptr)
     Expected union data *, found int *.
>>        func1(&t);     /* BAD_PARM */
[badparm3.c:28] **BAD_PARM(union)**
     Wrong type passed to func2 (argument 1: p)
     Expected int *, found union data *.
>>        func2(&u);     /* BAD_PARM */
```

- Source lines at which problems were detected.
- Description of the problem and the arguments that are incorrect.

Repair

Most of these problems are simple to correct based on the information given.

For example, problem #1 can be corrected by simply changing the incorrect line of code as follows

```
badparm1.c, line 6:if(strchr("testing", ’s’))
```

The other problems can be similarly corrected.

If your application generates error messages that you wish to ignore, you can add the option

```
insure++.suppress BAD_PARM
```

to your `.psrc` file.

This directive suppresses all BAD_PARM messages. If you wish to be more selective and suppress only a certain type of error, you can use the syntax

```
insure++.suppress BAD_PARM(class1, class2, …)
```

where the arguments are one or more of the identifiers for the various categories of error described on page 200. Similarly, you can enable suppressed error messages with the unsuppress option.

Thus, you could enable warnings about conflicts between types int and long (on systems where they are the same number of bytes) using the option

```
insure++.unsuppress BAD_PARM(compatible)
```

In addition to the keywords described on page 200, you can also use the type pointer to suppress all messages about different pointer types.

For example, many programs declare functions with the argument type char *, which are then called with pointers to various other data types. The ANSI standard recommends that you use type void * in such circumstances, since this is allowed to match any pointer type. If, for some reason, you cannot do this, you can suppress messages from Insure++ about incompatible pointer types with the option

```
insure++.suppress BAD_PARM(pointer)
```

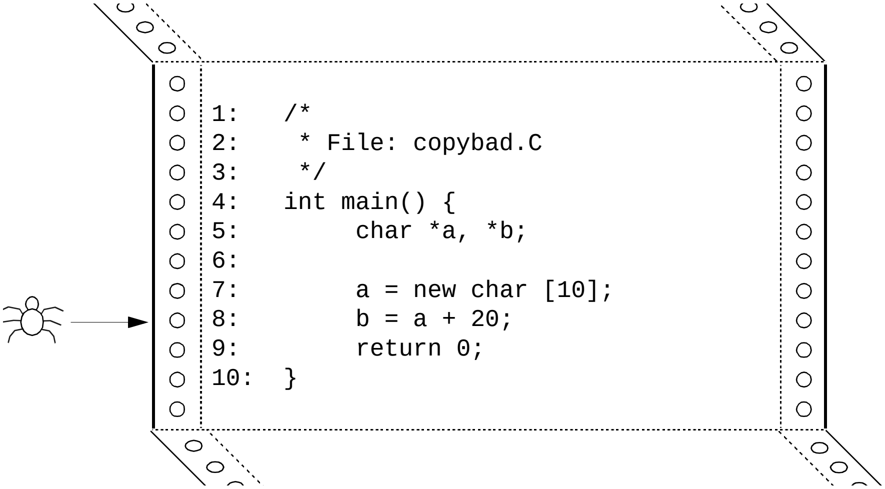
COPY_BAD_RANGE

Copying pointer which is out-of-range

This error is generated whenever an attempt is made to copy a pointer which points outside a valid range. It is not necessarily a serious problem, but may indicate faulty logic in the coding. Therefore, this error is suppressed by default.

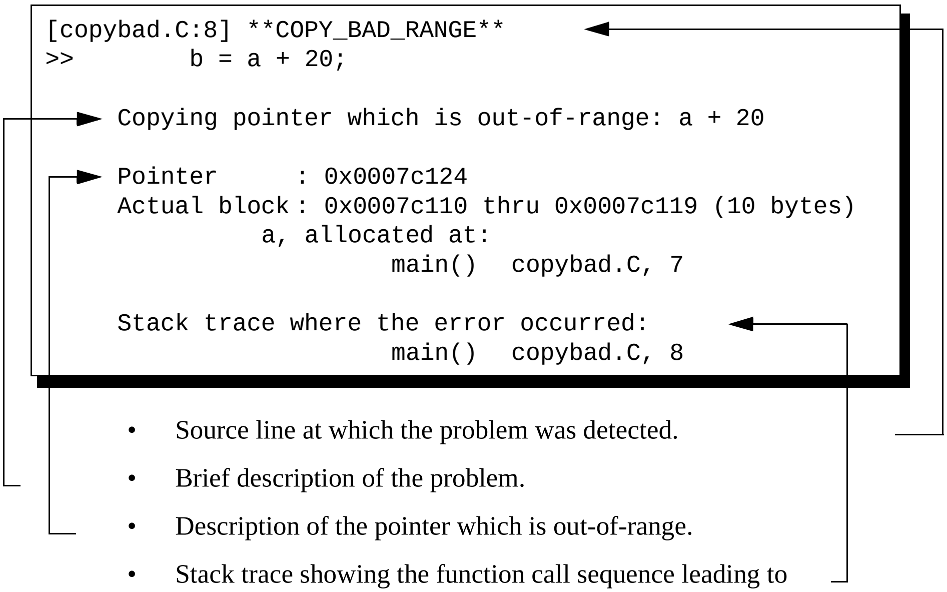
Problem

The following code illustrates the problem in a simple way. In line 7, the pointer `a` is initialized as an array of 10 `char`s. The next line then attempts to make pointer `b` point to an area which has not been allocated. The resulting pointer is not valid.



```
1:   /*
2:    * File: copybad.C
3:    */
4:   int main() {
5:        char *a, *b;
6:
7:        a = new char [10];
8:        b = a + 20;
9:        return 0;
10:  }
```

Diagnosis (at runtime)



```
[copybad.C:8] **COPY_BAD_RANGE**
>>          b = a + 20;

      Copying pointer which is out-of-range: a + 20

      Pointer      : 0x0007c124
      Actual block : 0x0007c110 thru 0x0007c119 (10 bytes)
                a, allocated at:
                         main()   copybad.C, 7

      Stack trace where the error occurred:
                         main()   copybad.C, 8
```

- Source line at which the problem was detected.
- Brief description of the problem.
- Description of the pointer which is out-of-range.
- Stack trace showing the function call sequence leading to the error.

Repair

The simple way to avoid this problem is to not copy the invalid pointer. There may be an incorrect boundary case in your code causing the problem.

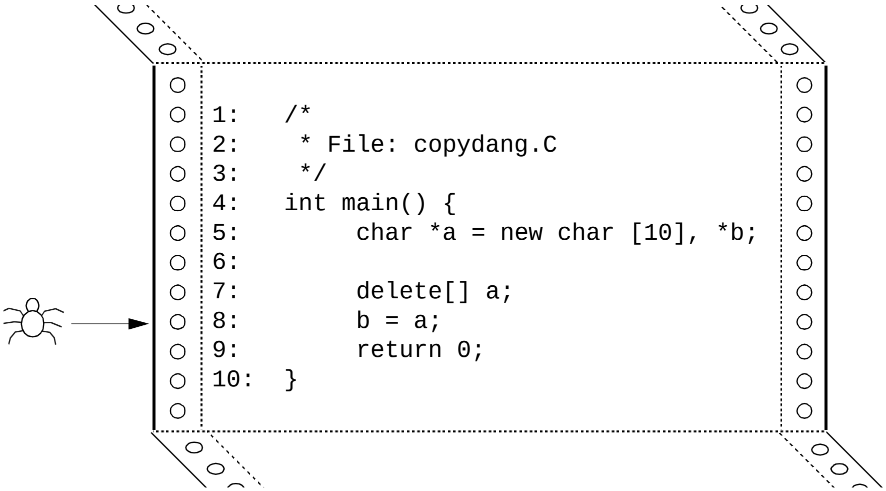
COPY_DANGLING

Copying pointer which has already been freed

This error is generated whenever an attempt is made to copy a pointer to a block of memory which has been freed. This error is suppressed by default.

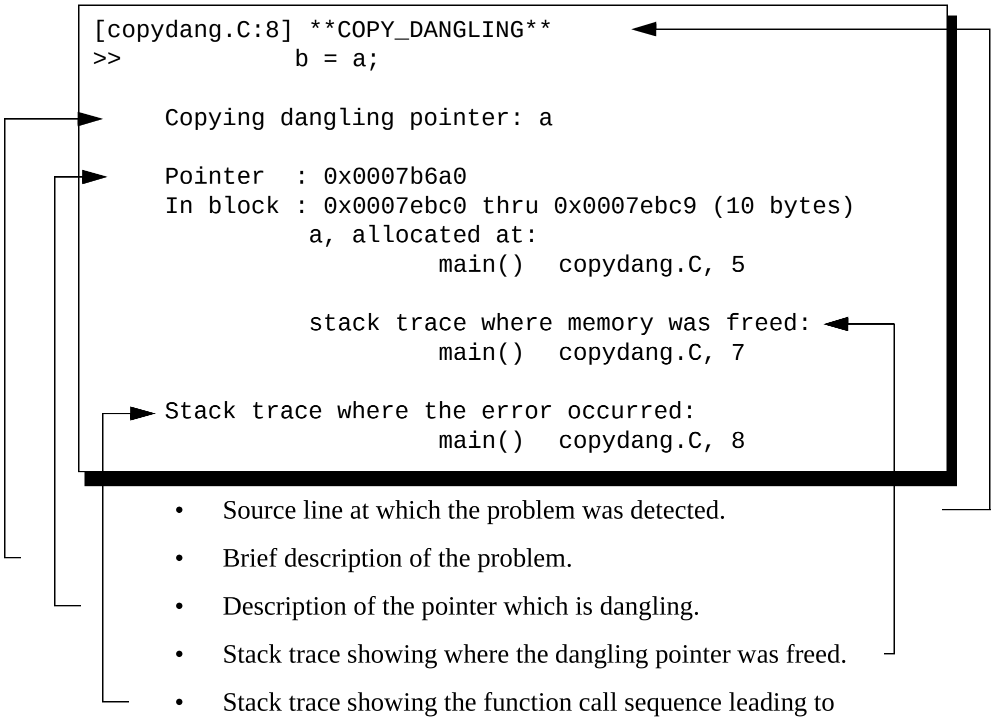
Problem

The following code illustrates the problem in a simple way. In line 7, the pointer `a` is freed by calling `delete[]`. The next line then attempts to copy from the address `a` into the variable `b`. Since `a` has already been freed, `b` will not point to valid memory either.



```
1:   /*
2:    * File: copydang.C
3:    */
4:   int main() {
5:        char *a = new char [10], *b;
6:
7:        delete[] a;
8:        b = a;
9:        return 0;
10:  }
```

Diagnosis (at runtime)



```
[copydang.C:8] **COPY_DANGLING**
>>           b = a;

     Copying dangling pointer: a

     Pointer  : 0x0007b6a0
     In block : 0x0007ebc0 thru 0x0007ebc9 (10 bytes)
               a, allocated at:
                        main()  copydang.C, 5

               stack trace where memory was freed:
                        main()  copydang.C, 7

     Stack trace where the error occurred:
                        main()  copydang.C, 8
```

- Source line at which the problem was detected.
- Brief description of the problem.
- Description of the pointer which is dangling.
- Stack trace showing where the dangling pointer was freed.
- Stack trace showing the function call sequence leading to the error.

Repair

The simple way to avoid this problem is to not attempt to reuse pointers after they have been freed. Check that the deallocation that occurs at the indicated location should have taken place. Also check that the pointer you are (mis)using should really be pointing to a block allocated at the indicated place.

COPY_UNINIT_PTR

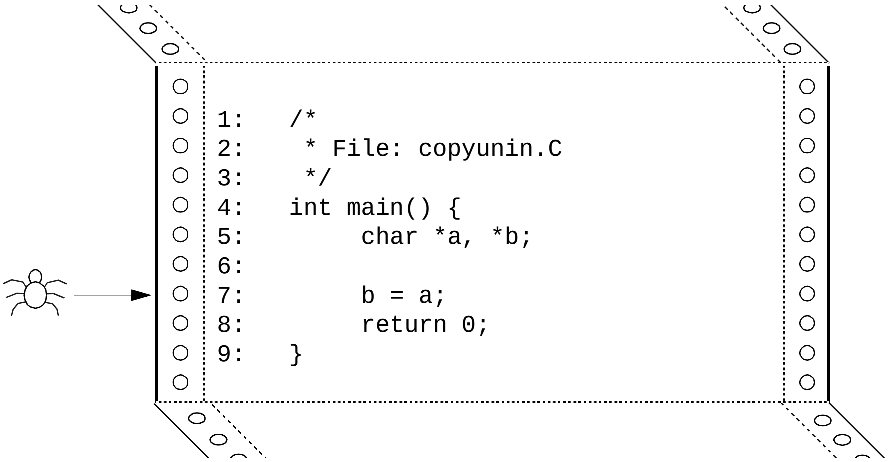
Copying uninitialized pointer

This error is generated whenever an uninitialized pointer is copied.

This error category will be disabled if full uninitialized memory checking is in effect (the default). In this case, errors are detected in the READ_UNINIT_MEM category instead.

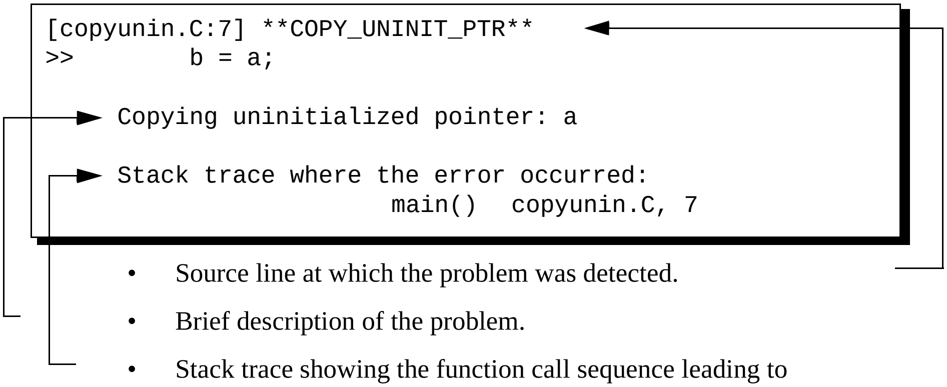
Problem

The pointer a is declared in line 5, but is never initialized. Therefore, when an attempt is made in line 7 to copy this pointer to b, an error is generated.



```
1:   /*
2:    * File: copyunin.C
3:    */
4:   int main() {
5:        char *a, *b;
6:
7:        b = a;
8:        return 0;
9:   }
```

Diagnosis (at runtime)



```
[copyunin.C:7] **COPY_UNINIT_PTR**
>>        b = a;

    Copying uninitialized pointer: a

    Stack trace where the error occurred:
                       main()  copyunin.C, 7
```

- Source line at which the problem was detected.
- Brief description of the problem.
- Stack trace showing the function call sequence leading to the error.

Repair

This problem is usually caused by omitting an assignment or allocation statement that would initialize a pointer. The example code given could be corrected by assigning a value to `a` before reaching line 7.

COPY_WILD

Copying wild pointer

This problem occurs when an attempt is made to copy a pointer whose value is invalid or which Insure++ did not see allocated.

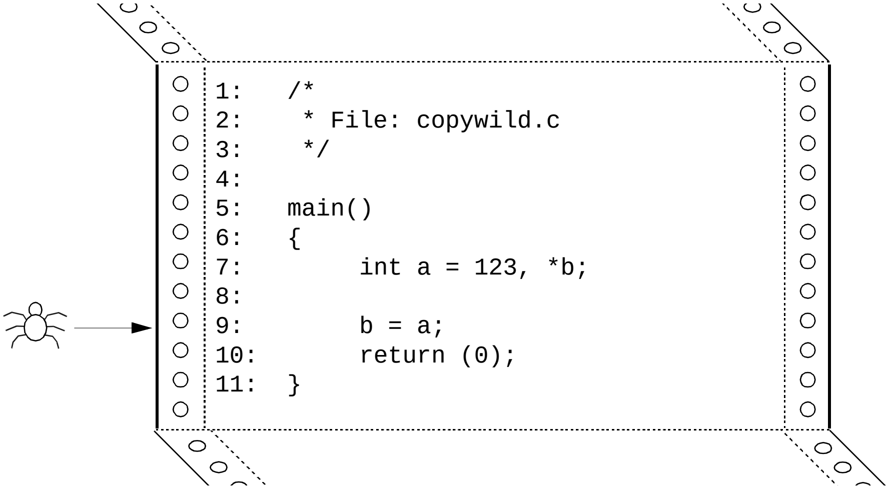
This can come about in several ways:

- Errors in user code that result in pointers that don't point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

This discussion centers on the first type of problem described here. A detailed discussion of the second topic, including samples of its generation and repair can be found in "Interfaces" on page 103.

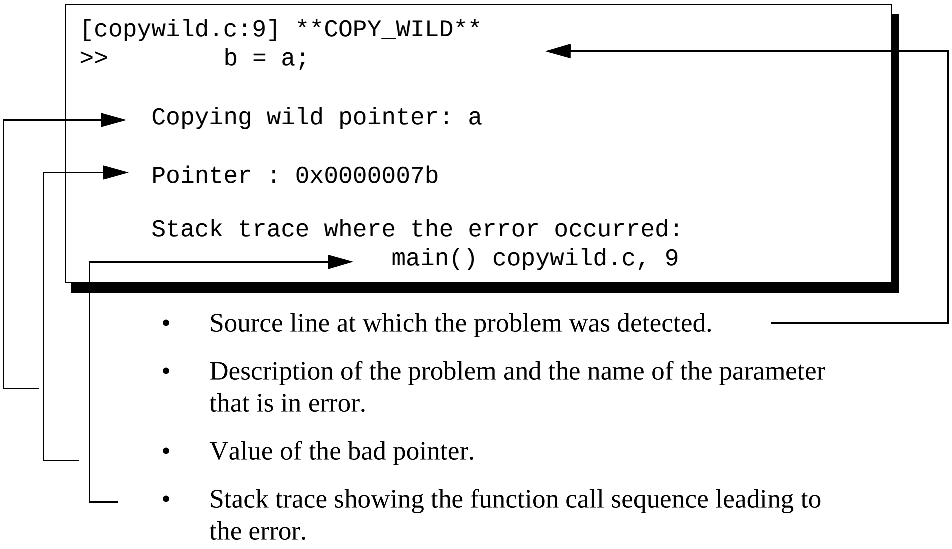
Problem

The following code attempts to use the address of a variable but contains an error at line 9 - the address operator (&) has been omitted.



```
1:   /*
2:     * File: copywild.c
3:     */
4:
5:   main()
6:   {
7:        int a = 123, *b;
8:
9:        b = a;
10:       return (0);
11:  }
```

Diagnosis (at runtime)



```
[copywild.c:9] **COPY_WILD**
>>       b = a;

     Copying wild pointer: a

     Pointer : 0x0000007b

     Stack trace where the error occurred:
                      main() copywild.c, 9
```

- Source line at which the problem was detected.
- Description of the problem and the name of the parameter that is in error.
- Value of the bad pointer.
- Stack trace showing the function call sequence leading to the error.

Note that most compilers will generate warning messages for this error since the assignment uses incompatible types.

DEAD_CODE

Code is not executed

This error is generated when code is not evaluated, has no effect, or is unreachable.

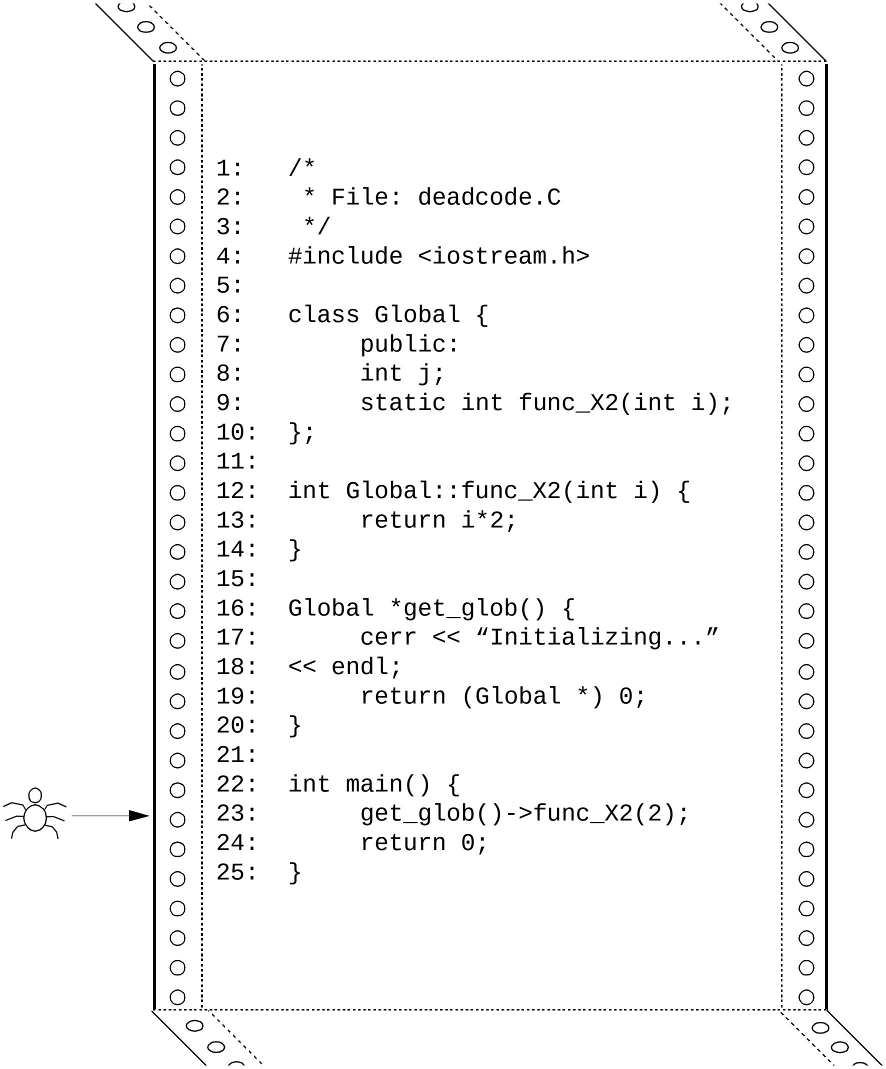
Insure++ distinguishes between several types of dead code as follows:

`emptystmt`	The statement is empty.
`emptyloopbody`	Loop body is empty.
`noeffect`	Code has no effect.
`notevaluated`	Code is not evaluated.

Error messages are classified according to this scheme and can be selectively enabled or disabled. By default, this error category is suppressed.

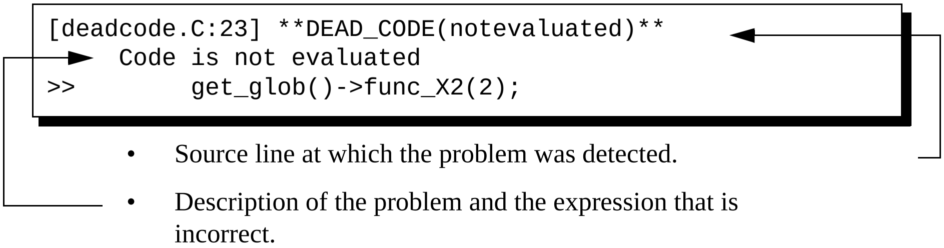
Problem #1

The following code shows a very tricky, well-disguised error that demonstrates how hard it is to find problems of this type without Insure++. The initialization function `get_glob` is never called by this code. Because `func_X2` is declared as a static function in the `Global` class, it can be called directly by `main`. This is in fact what happens, meaning that line 23 is interpreted as only a call to `func_X2`. Therefore, an error is generated since the call to `get_glob` is never evaluated.



```
/*
 * File: deadcode.C
 */
#include <iostream.h>

class Global {
     public:
     int j;
     static int func_X2(int i);
};

int Global::func_X2(int i) {
     return i*2;
}

Global *get_glob() {
     cerr << “Initializing...”
<< endl;
     return (Global *) 0;
}

int main() {
     get_glob()->func_X2(2);
     return 0;
}
```

Diagnosis (during compilation)



```
[deadcode.C:23] **DEAD_CODE(notevaluated)**
     Code is not evaluated
>>        get_glob()->func_X2(2);
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is incorrect.

Repair

This problem can be solved by replacing line 23 with a direct call to `func_X2`:

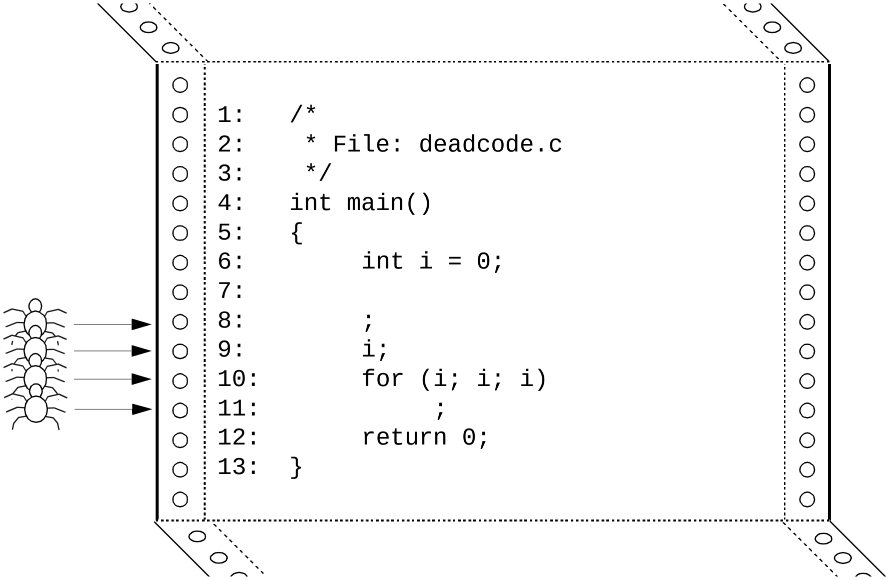
```
Global::func_X2(2);
```

or by not making `func_X2` a static function.

In some cases, it may be that the dead code was never intended to be called. If that is the case, the dead code should be eliminated for clarity.

Problem #2

The following code illustrates several other types of DEAD_CODE errors, this time in C.



```
/*
 * File: deadcode.c
 */
int main()
{
     int i = 0;

     ;
     i;
     for (i; i; i)
          ;
     return 0;
}
```

Diagnosis (during compilation)

```
[deadcode.c:8] **DEAD_CODE(emptystmt)**
     Statement is empty
>>        ;
[deadcode.c:9] **DEAD_CODE(noeffect)**
     Code has no effect
>>        i;
[deadcode.c:10] **DEAD_CODE(noeffect)**
     For loop initializer has no effect
>>        for (i; i; i)
[deadcode.c:10] **DEAD_CODE(noeffect)**
     For loop increment has no effect
>>        for (i; i; i)
[deadcode.c:11] **DEAD_CODE(emptyloopbody)**
     Loop body is empty (may be okay)
>>        ;
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is incorrect.

Repair

These errors are usually corrected by removing the superfluous statement or by modifying the statement so that it does what it was intended to do, e.g add a missing increment operator. An empty loop body may be useful in certain situations. In such a case, you may want to suppress that subcategory of DEAD_CODE.

DELETE_MISMATCH

Inconsistent usage of delete operator

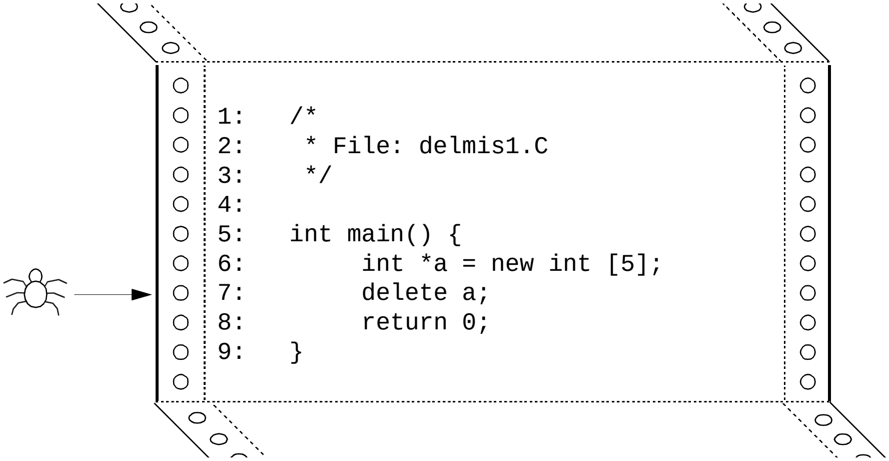
Error Codes

The current version of ANSI C++ distinguishes between memory allocated with `new` and `new[]`. A `delete` call *must* (according to the standard) match the `new` call, i.e. whether or not it has `[]`.

Calling `new[]` and `delete` may cause the compiler to not call the destructor on each element of the array, which can lead to serious errors. Even worse, if the memory was allocated differently, memory may be corrupted. This is definitely poor practice and unlikely to work with future releases of the specific compiler.

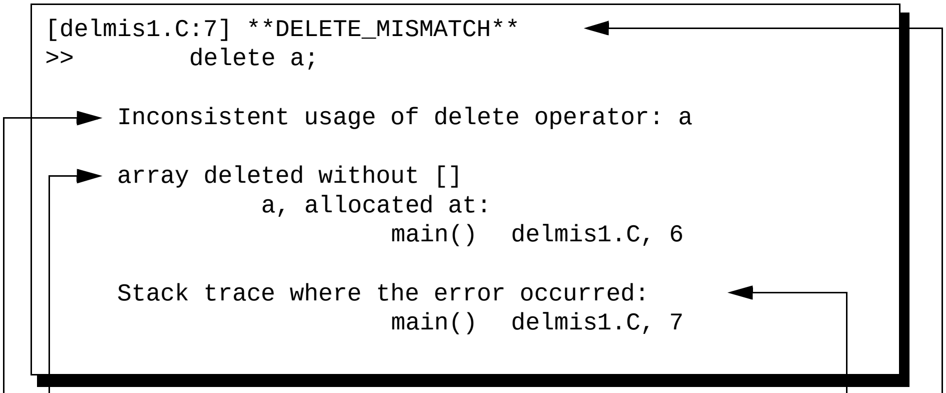
Problem #1

The following code shows a block of memory allocated with `new[]` and freed with `delete`, without `[]`.



```
/*
 * File: delmis1.C
 */

int main() {
     int *a = new int [5];
     delete a;
     return 0;
}
```

Diagnosis (at runtime)

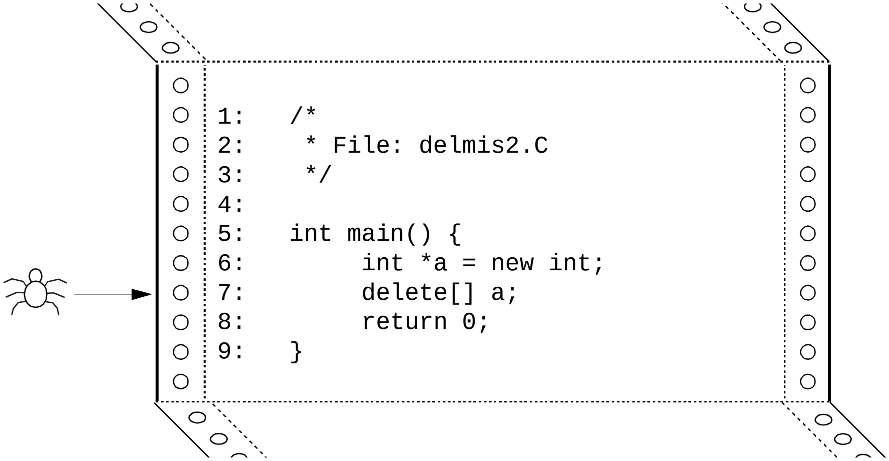


```
[delmis1.C:7] **DELETE_MISMATCH**
>>        delete a;

     Inconsistent usage of delete operator: a

     array deleted without []
               a, allocated at:
                        main()  delmis1.C, 6

     Stack trace where the error occurred:
                        main()  delmis1.C, 7
```

- Source line at which the problem was detected.
- Description of the problem and the operator which doesn’t match.
- Brief description of the mismatch.
- Stack trace showing the function call sequence leading to the error.

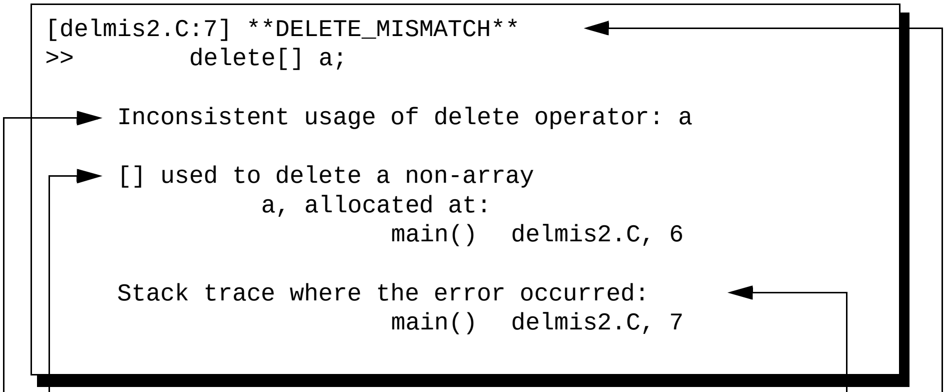
Problem #2

The following code shows a block of memory allocated with `new`, without `[]`, and freed with `delete[]`. This may cause some implementations of C++ to crash, because the compiler may look for extra bits of information about how the block was allocated. Some compilers do this to allow this type of error, extending the ANSI standard. In this case, there would be no extra bits, so the compiler would attempt to read from an invalid memory address.



```
/*
 * File: delmis2.C
 */

int main() {
     int *a = new int;
     delete[] a;
     return 0;
}
```

Diagnosis (at runtime)



```
[delmis2.C:7] **DELETE_MISMATCH**
>>        delete[] a;

     Inconsistent usage of delete operator: a

     [] used to delete a non-array
               a, allocated at:
                        main()  delmis2.C, 6

     Stack trace where the error occurred:
                        main()  delmis2.C, 7
```

- Source line at which the problem was detected.
- Description of the problem and the operator which doesn’t match.
- Brief description of the mismatch.
- Stack trace showing the function call sequence leading to the error.

Repair

To eliminate this error, you need to change the `delete` call to match the `new` call. In our first example, this could be accomplished by calling `delete[]` instead of `delete`, and vice versa in the second example.

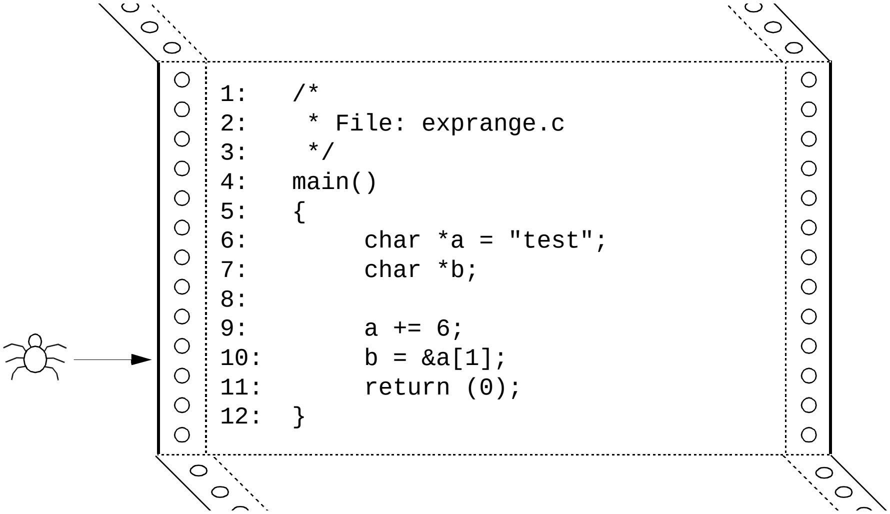
EXPR_BAD_RANGE

Expression exceeded range

This error is generated whenever an expression uses a pointer that is outside its legal range. In many circumstances, these pointers are then turned into legal values before use (e.g., code generated by automated programming tools such as `lex` and `yacc`), so this error category is suppressed by default. If used with their illegal values, other Insure++ errors will be displayed which can be tracked to their source by re-enabling this error class.

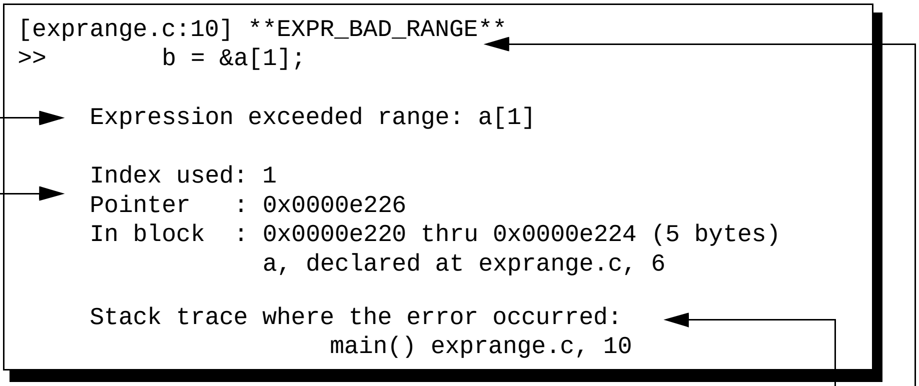
Problem

In this code, the pointer `a` initially points to a character string. It is subsequently incremented beyond the end of the string. When the resulting pointer is used to make an array reference, a range error is generated.



```
/*
 * File: exprange.c
 */
main()
{
     char *a = "test";
     char *b;

     a += 6;
     b = &a[1];
     return (0);
}
```

Diagnosis (at runtime)



```
[exprange.c:10] **EXPR_BAD_RANGE**
>>        b = &a[1];

     Expression exceeded range: a[1]

     Index used: 1
     Pointer   : 0x0000e226
     In block  : 0x0000e220 thru 0x0000e224 (5 bytes)
                 a, declared at exprange.c, 6

     Stack trace where the error occurred:
                      main() exprange.c, 10
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Description of the memory block to which the out of range pointer used to point, including the location at which it is declared.
- Stack trace showing the function call sequence leading to the error.

Repair

In most cases, this error is caused by incorrect logic in the code immediately prior to that at which the message is generated. Probably the simplest method of solution is to run the program under a debugger with a breakpoint at the indicated location.

If you cannot find the error by examining the values of other variables at this location, the program should be run again, stopped somewhere shortly before the indicated line, and single-stepped until the problem occurs.

EXPR_DANGLING

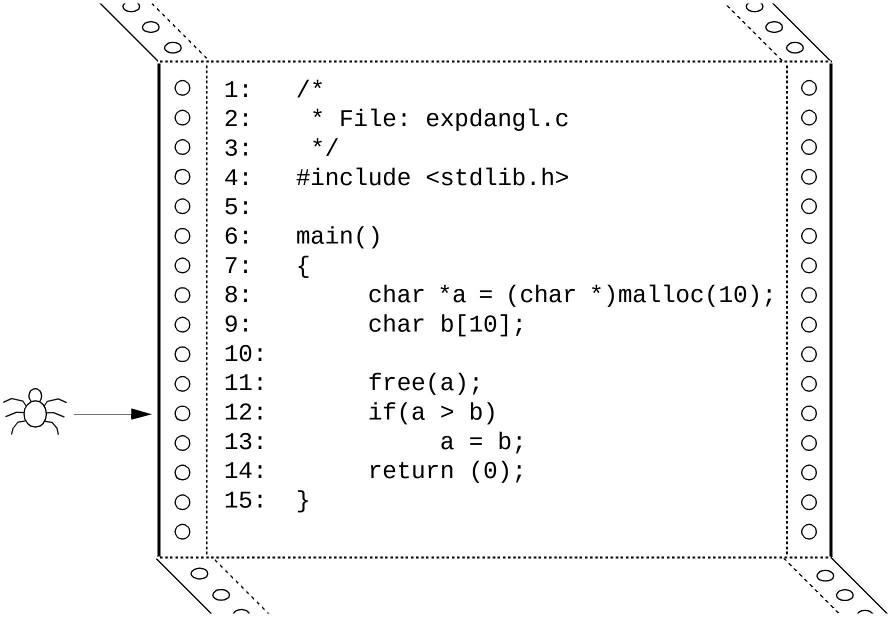
Expression uses dangling pointer

This error is generated whenever an expression operates on a dangling pointer - i.e., one which points to either

- A block of dynamically allocated memory that has already been freed.
- A block of memory which was allocated on the stack in some routine that has subsequently returned.

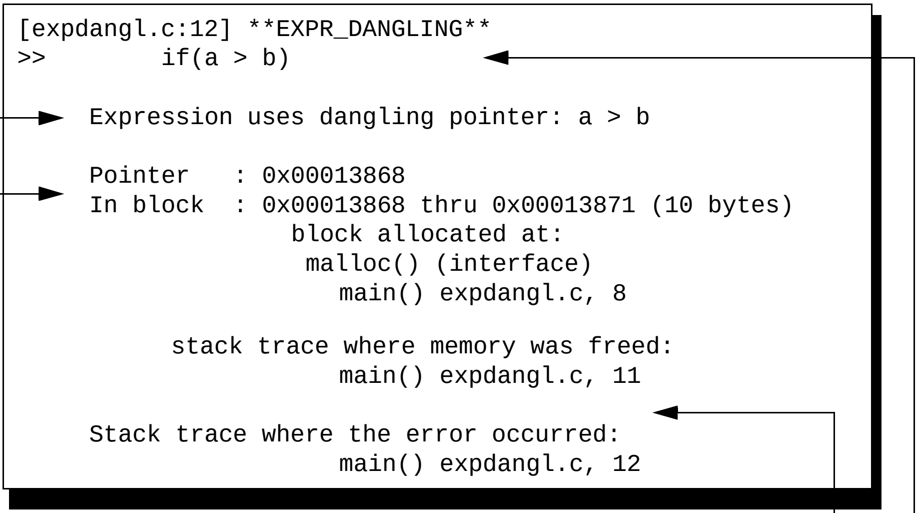
Problem

The following code fragment shows a block of memory being allocated and then freed. After the memory is de-allocated, the pointer to it is used again, even though it no longer points to valid memory.



```
/*
 * File: expdangl.c
 */
#include <stdlib.h>

main()
{
     char *a = (char *)malloc(10);
     char b[10];

     free(a);
     if(a > b)
          a = b;
     return (0);
}
```

Diagnosis (at runtime)



```
[expdangl.c:12] **EXPR_DANGLING**
>>          if(a > b)

     Expression uses dangling pointer: a > b

     Pointer   : 0x00013868
     In block  : 0x00013868 thru 0x00013871 (10 bytes)
                   block allocated at:
                    malloc() (interface)
                      main() expdangl.c, 8

           stack trace where memory was freed:
                      main() expdangl.c, 11

     Stack trace where the error occurred:
                      main() expdangl.c, 12
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Description of the memory block to which the pointer used to point, including the location at which it was allocated and subsequently freed.
- Stack trace showing the function call sequence leading to the error.

Repair

A good first check is to see if the pointer used in the expression at the indicated line is actually the one intended.

If it appears to be the correct pointer, check the line of code where the block was freed (as shown in the error message) to see if it was freed incorrectly.

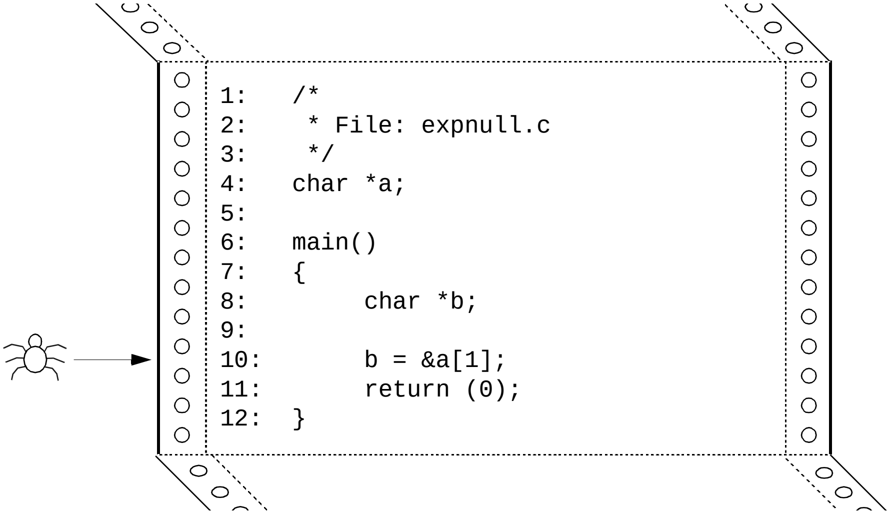
EXPR_NULL

Expression uses NULL pointer

This error is generated whenever an expression operates on the NULL pointer.

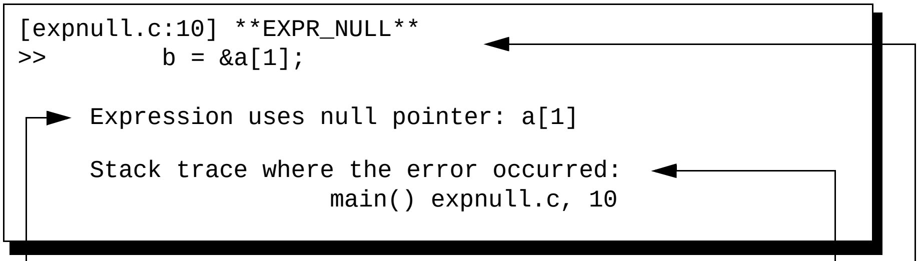
Problem

The following code fragment declares a pointer, a, which is initialized to zero by virtue of being a global variable. It then manipulates this pointer, generating the EXPR_NULL error.



```
/*
 * File: expnull.c
 */
char *a;

main()
{
     char *b;

     b = &a[1];
     return (0);
}
```

Diagnosis (at runtime)



```
[expnull.c:10] **EXPR_NULL**
>>        b = &a[1];

     Expression uses null pointer: a[1]

     Stack trace where the error occurred:
                     main() expnull.c, 10
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Stack trace showing the function call sequence leading to the error.

Repair

One potential cause of this error is shown in this example. The pointer a is a global variable that will be initialized to zero by the compiler. Since this variable is never modified to point to anything else, it is still NULL when first used.

One way the given code can be corrected is by adding an assignment as follows

```
/*
 * File: expnull.c (modified)
 */
char *a;
main()
{
      char *b, c[10];
      a = c;
      b = &a[1];
      return (0);
}
```

It could also be corrected by allocating a block of memory.

A second possibility is that the pointer was set to zero by the program at some point before its subsequent use and not re-initialized. This is common in programs which make heavy use of dynamically allocated memory and which mark freed blocks by resetting their pointers to NULL.

A final common problem is caused when one of the dynamic memory allocation routines, malloc, calloc, or realloc, fails and returns a NULL pointer. This can happen either because your program passes bad arguments or simply because it asks for too much memory. A simple way of finding this problem with Insure++ is to enable the RETURN_FAILURE error code (see page 342) via your .psrc file and run the program again. It will then issue diagnostic messages every time a system call fails, including the memory allocation routines.

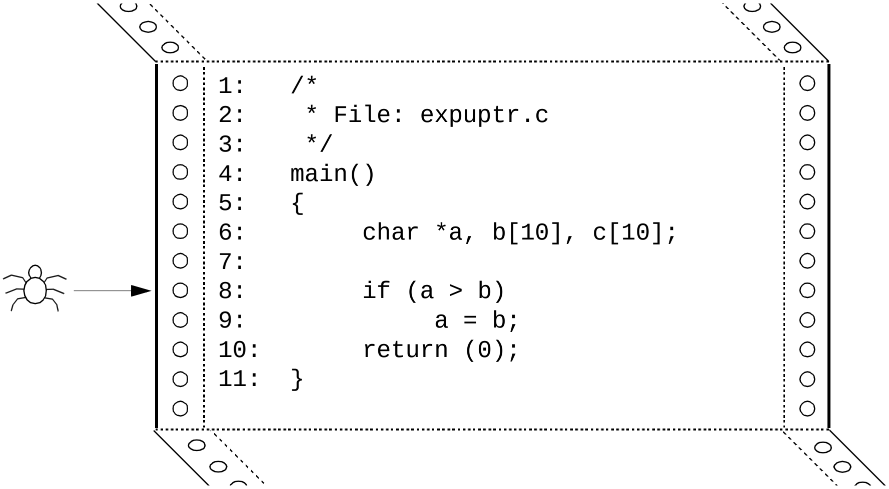
EXPR_UNINIT_PTR

Expression uses uninitialized pointer

This error is generated whenever an expression operates on an uninitialized pointer.

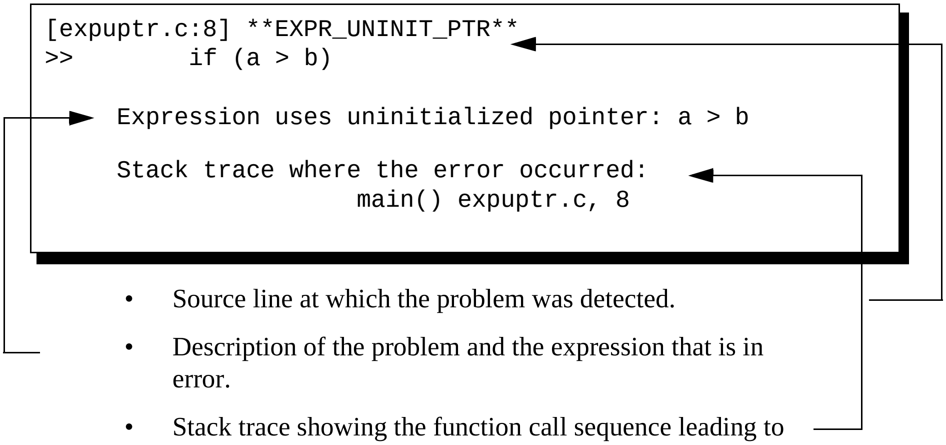
Problem

The following code uses an uninitialized pointer.



```
/*
 * File: expuptr.c
 */
main()
{
     char *a, b[10], c[10];

     if (a > b)
          a = b;
     return (0);
}
```

Diagnosis (at runtime)



```
[expuptr.c:8] **EXPR_UNINIT_PTR**
>>        if (a > b)

     Expression uses uninitialized pointer: a > b

     Stack trace where the error occurred:
                     main() expuptr.c, 8
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Stack trace showing the function call sequence leading to the error.

Repair

This error is normally caused by omitting an assignment statement for the uninitialized variable. The example code can be corrected as follows:

```
/*
 * File: expuptr.c (modified)
 */
main()
{
    char *a, b[10], c[10];

    a = c;
    if (a > b)
        a = b;
    return (0);
}
```

EXPR_UNRELATED_PTRCMP

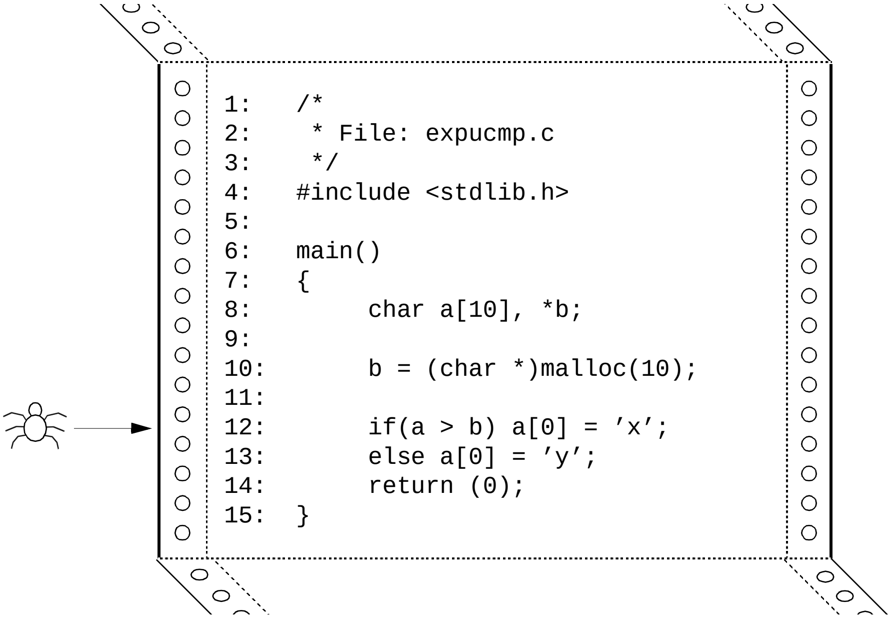
Expression compares unrelated pointers

This error is generated whenever an expression tries to compare pointers that do not point into the same memory block. This only applies to the operators >, >=, <, and <=. The operators == and != are exempt from this case.

The ANSI C-language specification declares this construct undefined except in the special case where a pointer points to an address one past the end of a block.

Problem

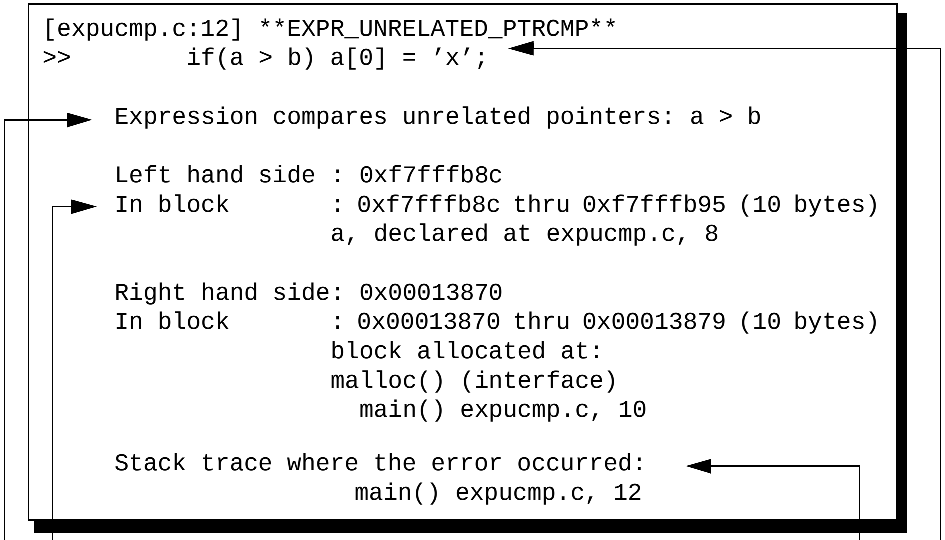
The following code illustrates the problem by comparing pointers to two data objects.



```
1:   /*
2:    * File: expucmp.c
3:    */
4:   #include <stdlib.h>
5:
6:   main()
7:   {
8:        char a[10], *b;
9:
10:       b = (char *)malloc(10);
11:
12:       if(a > b) a[0] = ’x’;
13:       else a[0] = ’y’;
14:       return (0);
15:  }
```

Note that the error in this code is not that the two objects `a` and `b` are of different data types (array vs. dynamic memory block), but that the comparison in line 12 attempts to compare pointers which do not point into the same memory block. According to the ANSI specification, this is an undefined operation.

Diagnosis (at runtime)



```
[expucmp.c:12] **EXPR_UNRELATED_PTRCMP**
>>          if(a > b) a[0] = 'x';

      Expression compares unrelated pointers: a > b

      Left hand side : 0xf7fffb8c
      In block       : 0xf7fffb8c thru 0xf7fffb95 (10 bytes)
                       a, declared at expucmp.c, 8

      Right hand side: 0x00013870
      In block       : 0x00013870 thru 0x00013879 (10 bytes)
                       block allocated at:
                       malloc() (interface)
                         main() expucmp.c, 10

      Stack trace where the error occurred:
                         main() expucmp.c, 12
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Description of the two pointers involved in the comparison. For each pointer, the associated block of memory is shown together with its size and the line number at which it was declared or allocated.
- Stack trace showing the function call sequence leading to the error.

Repair

While this construct is technically undefined according to the ANSI C specification, it is supported on many machines and its use is fairly common practice. If your application genuinely needs to use this construct, you can suppress error messages by adding the option

```
insure++.suppress EXPR_UNRELATED_PTRCMP
```

to your `.psrc` file.

EXPR_UNRELATED_PTRDIFF

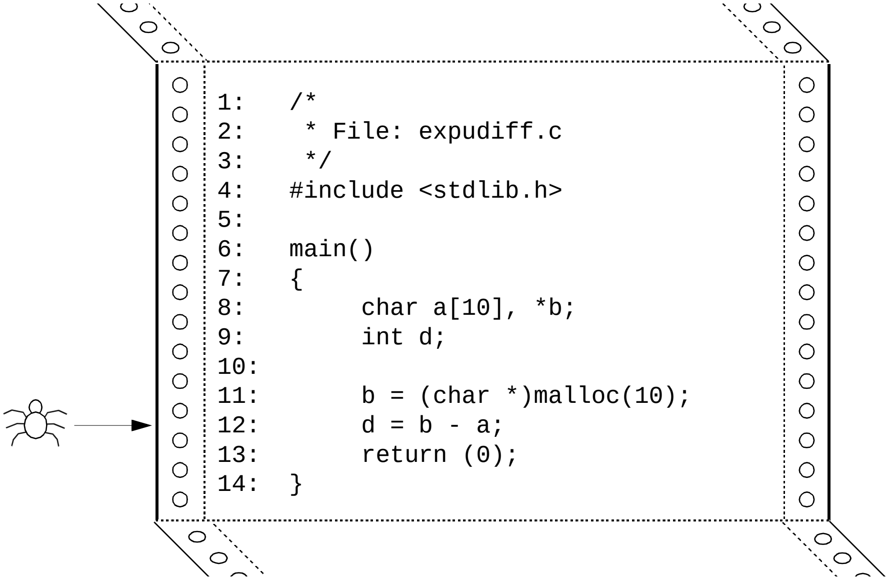
Expression subtracts unrelated pointers

This error is generated whenever an expression tries to compute the difference between pointers that do not point into the same memory block.

The ANSI C language specification declares this construct undefined except in the special case where a pointer points to an object one past the end of a block.

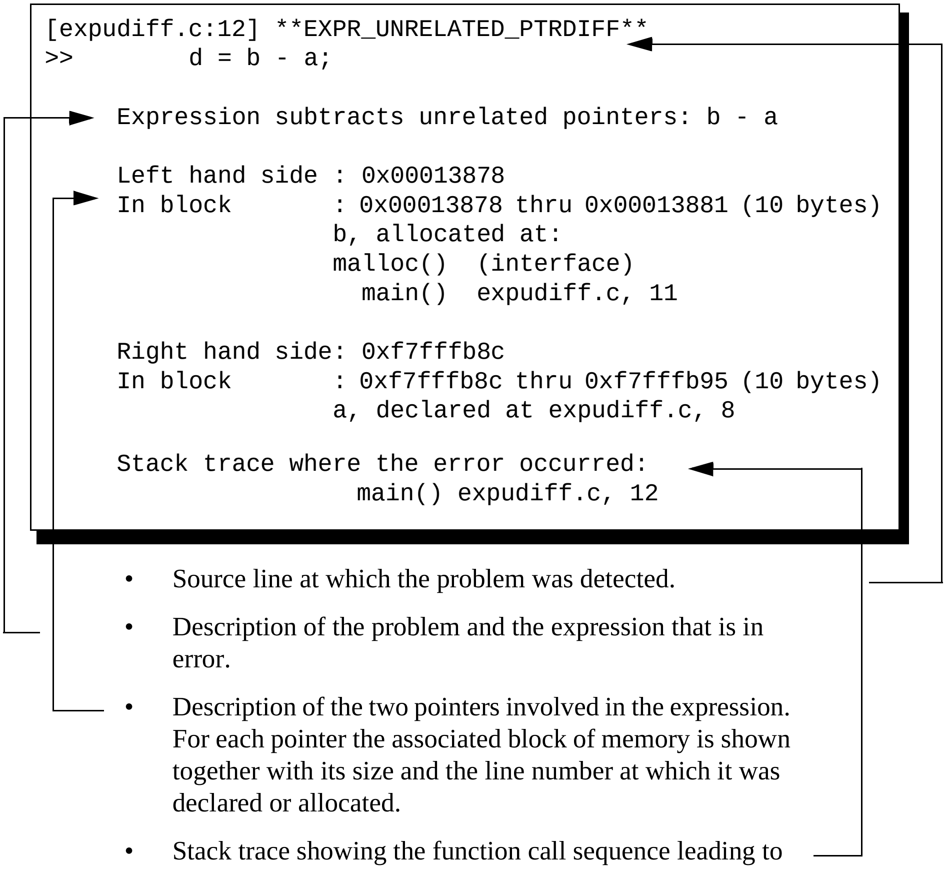
Problem

The following code illustrates the problem by subtracting two pointers to different data objects.



```
/*
 * File: expudiff.c
 */
#include <stdlib.h>

main()
{
     char a[10], *b;
     int d;

     b = (char *)malloc(10);
     d = b - a;
     return (0);
}
```

Diagnosis (at runtime)



```
[expudiff.c:12] **EXPR_UNRELATED_PTRDIFF**
>>        d = b - a;

     Expression subtracts unrelated pointers: b - a

     Left hand side : 0x00013878
     In block       : 0x00013878 thru 0x00013881 (10 bytes)
                    b, allocated at:
                    malloc()  (interface)
                      main()  expudiff.c, 11

     Right hand side: 0xf7fffb8c
     In block       : 0xf7fffb8c thru 0xf7fffb95 (10 bytes)
                    a, declared at expudiff.c, 8

     Stack trace where the error occurred:
                      main() expudiff.c, 12
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Description of the two pointers involved in the expression. For each pointer the associated block of memory is shown together with its size and the line number at which it was declared or allocated.
- Stack trace showing the function call sequence leading to the error.

Repair

While this construct is undefined according to the ANSI C language specification, it is supported on many machines and its use is fairly common practice. If your application genuinely needs to use this construct, you can suppress error messages by adding the option

```
        insure++.suppress EXPR_UNRELATED_PTRDIFF
```

to your .psrc file.

EXPR_WILD

Expression uses wild pointer

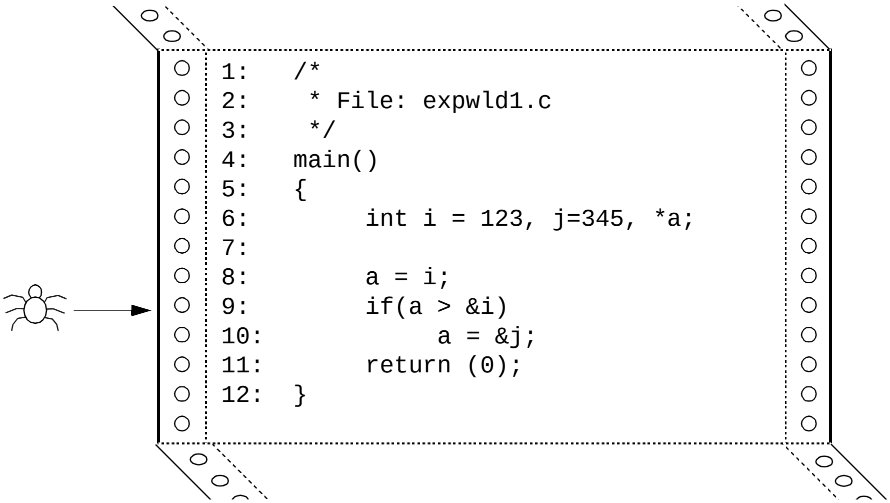
This error is generated whenever a program operates on a memory region that is unknown to Insure++. This can come about in two ways:

- Errors in user code that result in pointers that don't point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

This discussion centers on the first type of problem described here. A detailed discussion of the second topic, including samples of its generation and repair can be found in "Interfaces" on page 103.

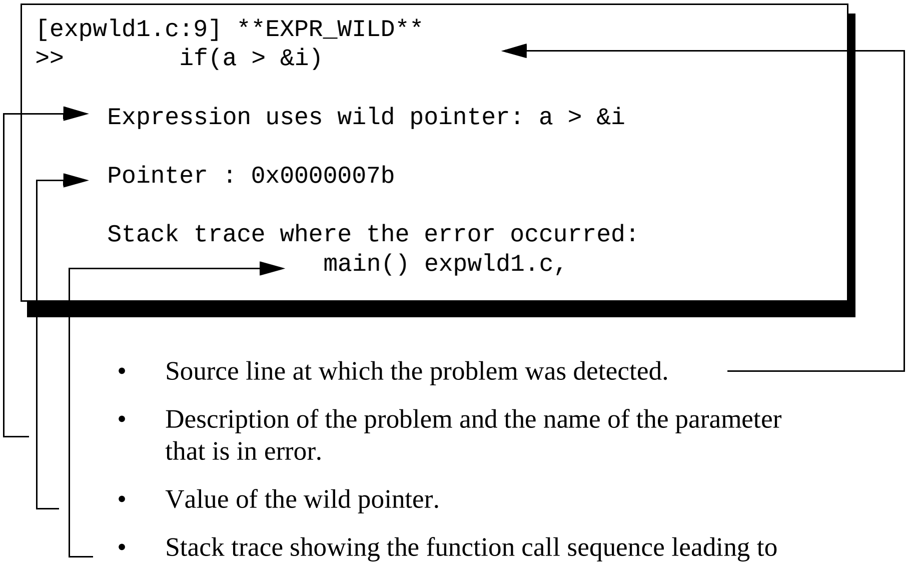
Problem #1

The following code attempts to use the address of a local variable but contains an error at line 8 - the address operator (&) has been omitted.



```
1:   /*
2:    * File: expwld1.c
3:    */
4:   main()
5:   {
6:        int i = 123, j=345, *a;
7:
8:        a = i;
9:        if(a > &i)
10:            a = &j;
11:       return (0);
12:  }
```

Diagnosis (at runtime)



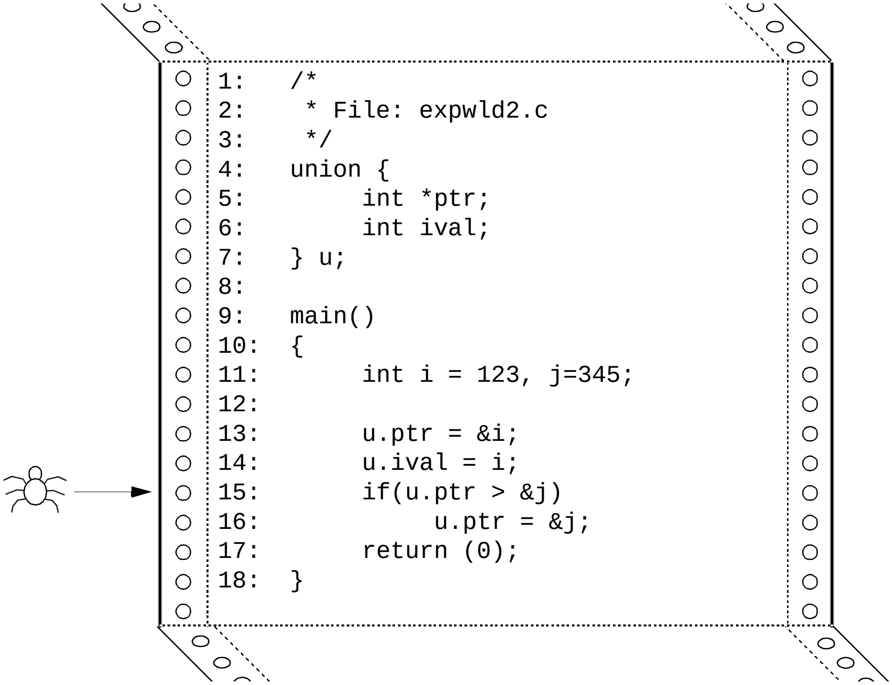
```
[expwld1.c:9] **EXPR_WILD**
>>        if(a > &i)

     Expression uses wild pointer: a > &i

     Pointer : 0x0000007b

     Stack trace where the error occurred:
                    main() expwld1.c,
```

- Source line at which the problem was detected.
- Description of the problem and the name of the parameter that is in error.
- Value of the wild pointer.
- Stack trace showing the function call sequence leading to the error.

Note that most compilers will generate warning messages for this error since the assignment in line 8 uses incompatible types.

Problem #2

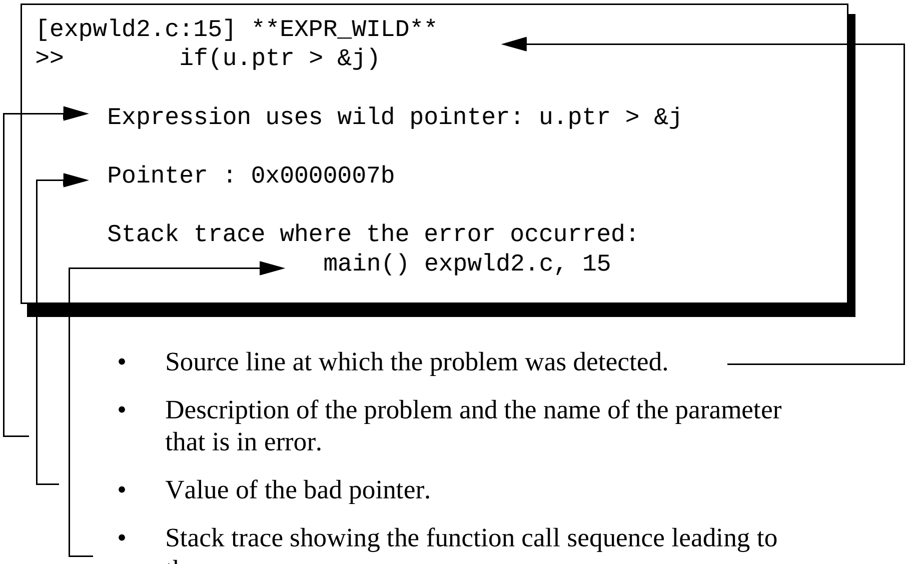
A more insidious version of the same problem can occur when using `union` types. The following code first assigns the pointer element of a union but then overwrites it with another element before finally attempting to use it.



```
/*
 * File: expwld2.c
 */
union {
     int *ptr;
     int ival;
} u;

main()
{
     int i = 123, j=345;

     u.ptr = &i;
     u.ival = i;
     if(u.ptr > &j)
          u.ptr = &j;
     return (0);
}
```

Note that this code will not generate compile time errors.

Diagnosis (at runtime)



```
[expwld2.c:15] **EXPR_WILD**
>>        if(u.ptr > &j)

     Expression uses wild pointer: u.ptr > &j

     Pointer : 0x0000007b

     Stack trace where the error occurred:
                    main() expwld2.c, 15
```

- Source line at which the problem was detected.
- Description of the problem and the name of the parameter that is in error.
- Value of the bad pointer.
- Stack trace showing the function call sequence leading to the error.

Repair

The simpler types of problem are most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively you can stop the program at some point prior to the error and single-step it through the code leading up to the error.

“Wild pointers” can also be generated when Insure++ has only partial information about your program’s structure. This issue is discussed extensively in “Interfaces” on page 103.

.

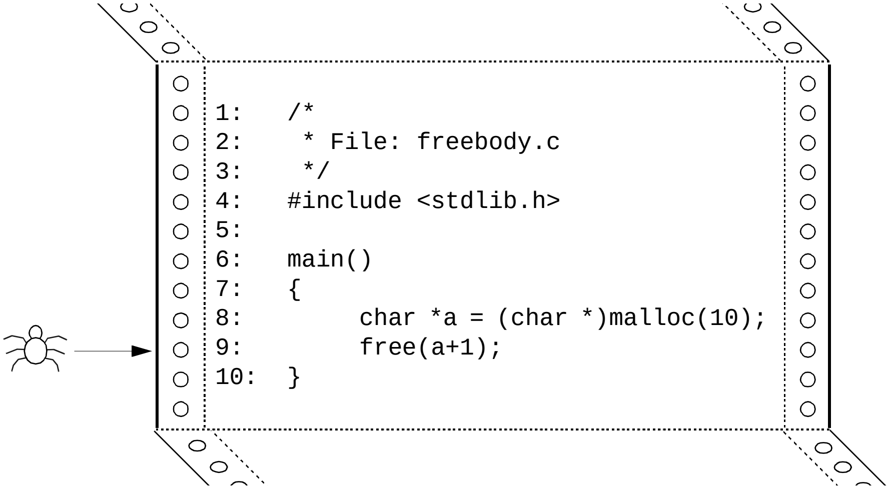
FREE_BODY

Freeing memory block from body

This error is generated when an attempt is made to de-allocate memory by using a pointer which currently points into the middle of a block, rather than to its beginning.

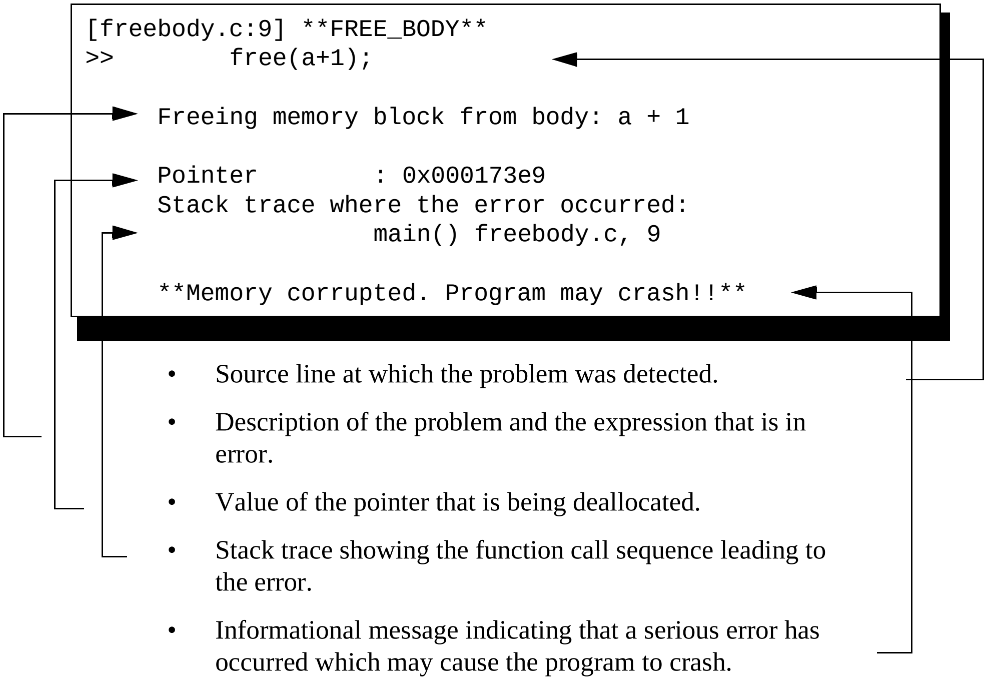
Problem

The following code attempts to free a memory region using an invalid pointer.



```
/*
 * File: freebody.c
 */
#include <stdlib.h>

main()
{
     char *a = (char *)malloc(10);
     free(a+1);
}
```

Diagnosis (at runtime)



```
[freebody.c:9] **FREE_BODY**
>>        free(a+1);

     Freeing memory block from body: a + 1

     Pointer        : 0x000173e9
     Stack trace where the error occurred:
                    main() freebody.c, 9

     **Memory corrupted. Program may crash!!**
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Value of the pointer that is being deallocated.
- Stack trace showing the function call sequence leading to the error.
- Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

This is normally a serious error. In most cases, the line number indicated in the diagnostics will have a simple error that can be corrected.

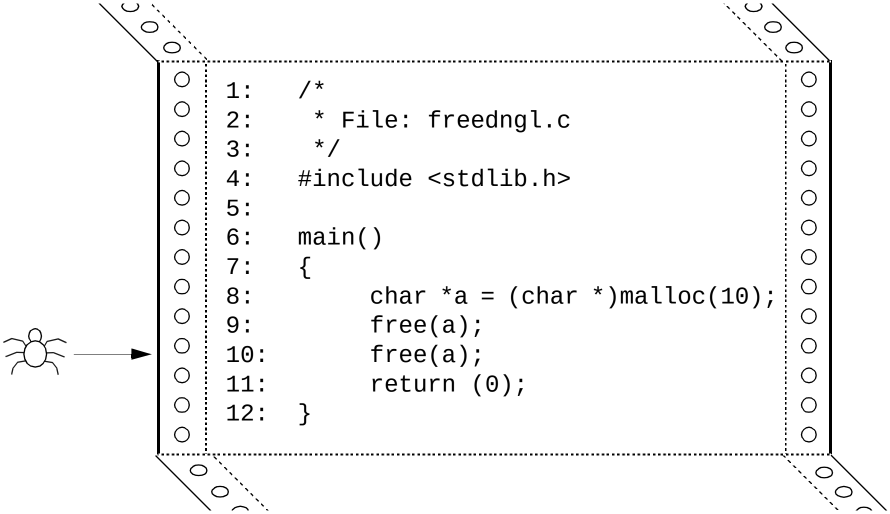
FREE_DANGLING

Freeing dangling pointer

This error is generated when a memory block is freed multiple times.

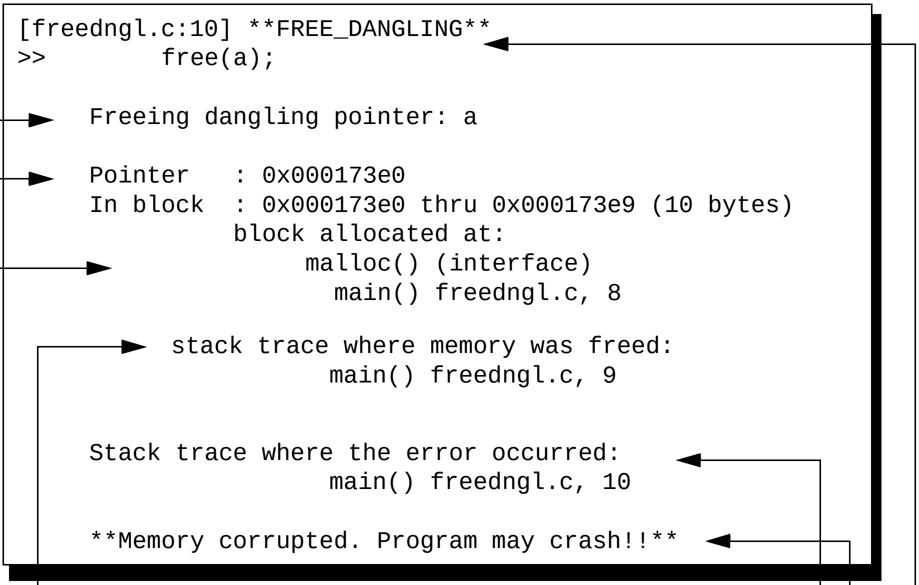
Problem

The following code frees the same pointer twice.



```
/*
 * File: freedngl.c
 */
#include <stdlib.h>

main()
{
     char *a = (char *)malloc(10);
     free(a);
     free(a);
     return (0);
}
```

Diagnosis (at runtime)



```
[freedngl.c:10] **FREE_DANGLING**
>>        free(a);

     Freeing dangling pointer: a

     Pointer   : 0x000173e0
     In block  : 0x000173e0 thru 0x000173e9 (10 bytes)
               block allocated at:
                    malloc() (interface)
                      main() freedngl.c, 8

           stack trace where memory was freed:
                     main() freedngl.c, 9


     Stack trace where the error occurred:
                     main() freedngl.c, 10

     **Memory corrupted. Program may crash!!**
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Value of the pointer that is being deallocated.
- Information about the block of memory addressed by this pointer, including information about where this block was allocated.
- Stack trace showing where this block was freed.
- Stack trace showing the function call sequence leading to the error.
- Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

Some systems allow memory blocks to be freed multiple times. However, this is not portable and is not a recommended practice.

The information supplied in the diagnostics will allow you to see the line of code which previously de-allocated this block of memory. You should attempt to remove one of the two calls.

If your application is unable to prevent multiple calls to deallocate the same block, you can suppress error messages by adding the option

```
      insure++.suppress FREE_DANGLING
```

to your .psrc file.

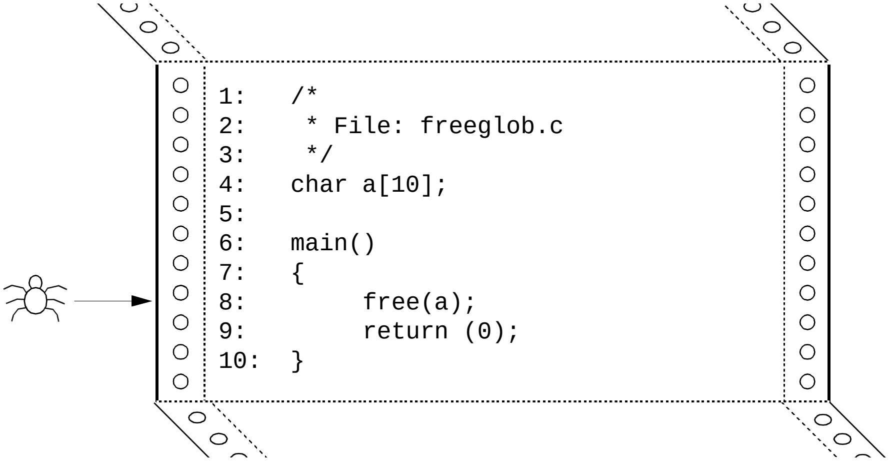
FREE_GLOBAL

Freeing global memory

This error is generated if the address of a global variable is passed to a routine that de-allocates memory.

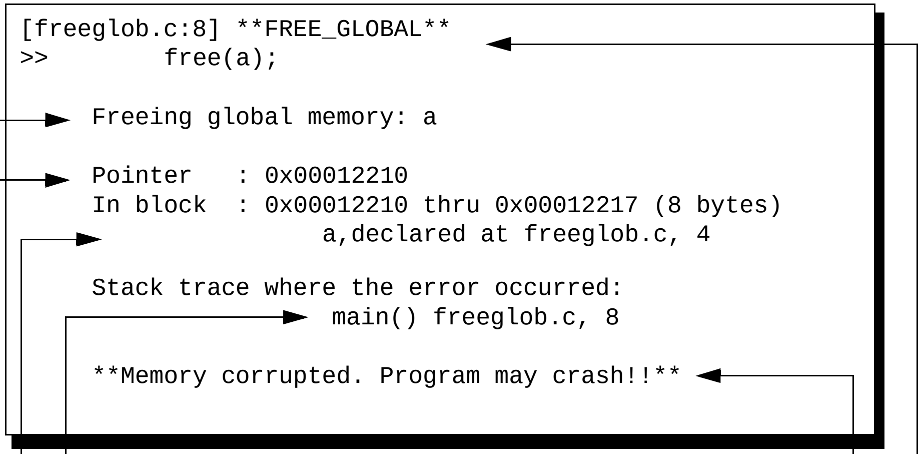
Problem

The following code attempts to deallocate a global variable that was not dynamically allocated.



```
/*
 * File: freeglob.c
 */
char a[10];

main()
{
     free(a);
     return (0);
}
```

Diagnosis (at runtime)



```
[freeglob.c:8] **FREE_GLOBAL**
>>        free(a);

     Freeing global memory: a

     Pointer   : 0x00012210
     In block  : 0x00012210 thru 0x00012217 (8 bytes)
                     a,declared at freeglob.c, 4

     Stack trace where the error occurred:
                      main() freeglob.c, 8

     **Memory corrupted. Program may crash!!**
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Value of the pointer that is being deallocated.
- Information about the block of memory addressed by this pointer, including information about where this block was declared.
- Stack trace showing the function call sequence leading to the error.
- Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

Some systems allow this operation, since they keep track of which blocks of memory are actually dynamically allocated, but this is not portable programming practice and is not recommended.

In some cases, this error will result from a simple coding mistake at the indicated source line which can be quickly corrected.

A more complex problem may arise when a program uses both statically and dynamically allocated blocks in the same way. A common example is a linked list in which the head of the list is static, while the other entries are allocated dynamically. In this case, you must take care not to free the static list head when removing entries.

If your application is unable to distinguish between global and dynamically allocated memory blocks, you can suppress error messages by adding the option

```
insure++.suppress FREE_GLOBAL
```

to your .psrc file.

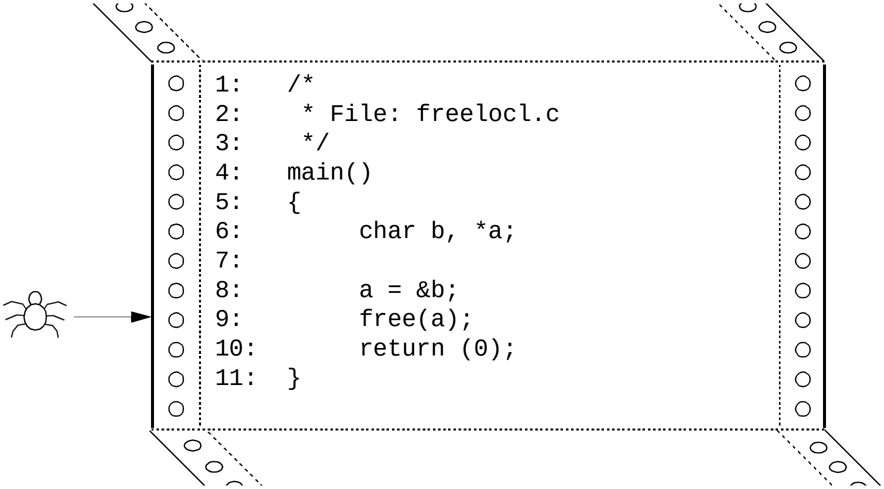
FREE_LOCAL

Freeing local memory

This error is generated if the address of a local variable is passed to `free`.

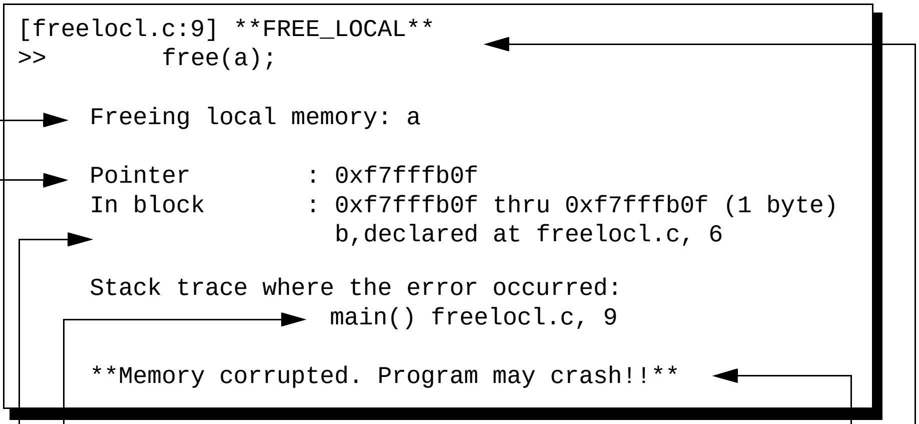
Problem

The following code attempts to free a local variable that was not dynamically allocated.



```
/*
 * File: freelocl.c
 */
main()
{
     char b, *a;

     a = &b;
     free(a);
     return (0);
}
```

Diagnosis (at runtime)



```
[freelocl.c:9] **FREE_LOCAL**
>>        free(a);

     Freeing local memory: a

     Pointer        : 0xf7fffb0f
     In block       : 0xf7fffb0f thru 0xf7fffb0f (1 byte)
                      b,declared at freelocl.c, 6

     Stack trace where the error occurred:
                      main() freelocl.c, 9

     **Memory corrupted. Program may crash!!**
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Value of the pointer that is being deallocated.
- Information about the block of memory addressed by this pointer, including information about where this block was declared.
- Stack trace showing the function call sequence leading to the error.
- Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

Some systems allow this operation since they keep track of which blocks of memory are actually dynamically allocated, but this is not portable programming practice and is not recommended.

In most cases, this error will result from a simple coding mistake at the indicated source line which can be quickly corrected.

If your application is unable to distinguish between local variables and dynamically allocated memory blocks, you can suppress error messages by adding the option

```
insure++.suppress FREE_LOCAL
```

to your .psrc file.

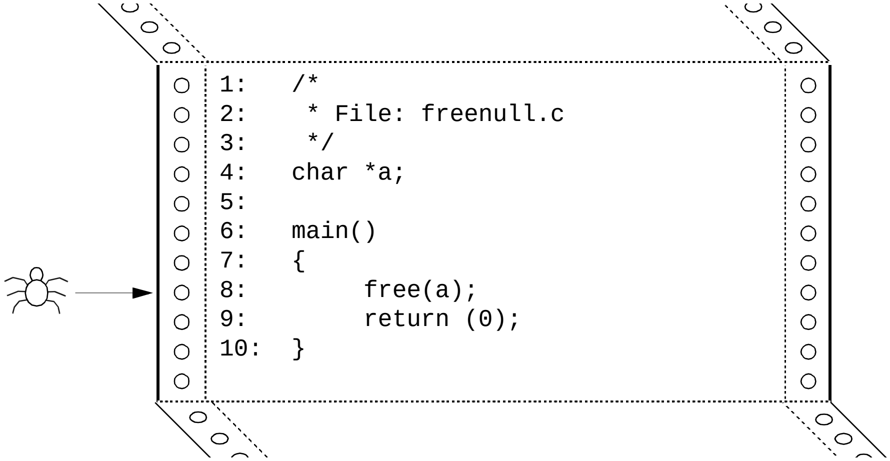
FREE_NULL

Freeing NULL pointer

This error is generated whenever an attempt is made to de-allocate memory using the NULL pointer.

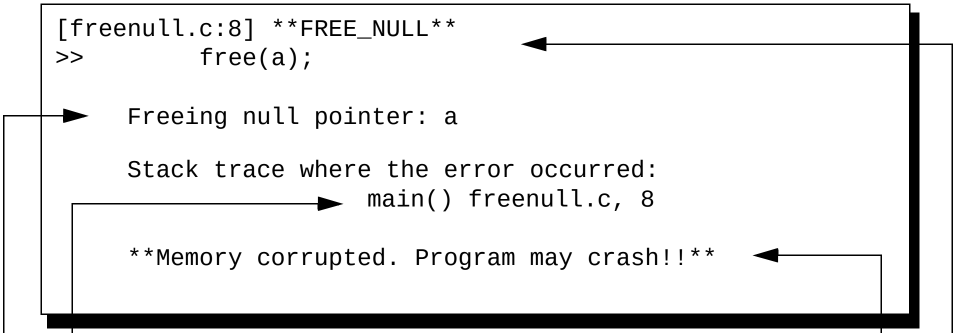
Problem

This code attempts to free the pointer a, which has never been assigned. Since this is a global variable, it is initialized to zero by default. This results in the code attempting to free a NULL pointer.



```
/*
 * File: freenull.c
 */
char *a;

main()
{
     free(a);
     return (0);
}
```

Diagnosis (at runtime)



```
[freenull.c:8] **FREE_NULL**
>>        free(a);

     Freeing null pointer: a

     Stack trace where the error occurred:
                     main() freenull.c, 8

     **Memory corrupted. Program may crash!!**
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Stack trace showing the function call sequence leading to the error.
- Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

Some systems allow this operation, but this is not portable programming practice and is not recommended.

A potential cause of this error is the one shown in the example - a pointer that never got explicitly initialized before being used. The given example can be corrected by adding an allocation as follows

```
/*
 * File: freenull.c (modified)
 */
#include <stdlib.h>
char *a;
```

```
main()
{
      a = (char *)malloc(100);
      free(a);
      return (0);
}
```

A second fairly common possibility is that a block of dynamically allocated memory associated with the pointer has already been freed, and its pointer reset to NULL. In this case, the error could mean that a second attempt is being made to free the same memory block.

A final common problem is caused when one of the dynamic memory allocation routines, malloc, calloc, or realloc, fails and returns a NULL pointer. This can happen either because your program passes bad arguments, or simply because it asks for too much memory. A simple way of finding this problem with Insure++ is to enable the RETURN_FAILURE error code (see page 342) via your .psrc file and run the program again. It will then issue diagnostic messages every time a system call fails, including the memory allocation routines.

If your application needs to free NULL pointers, you can suppress these error messages by adding the option

```
insure++.suppress FREE_NULL
```

to your .psrc file.

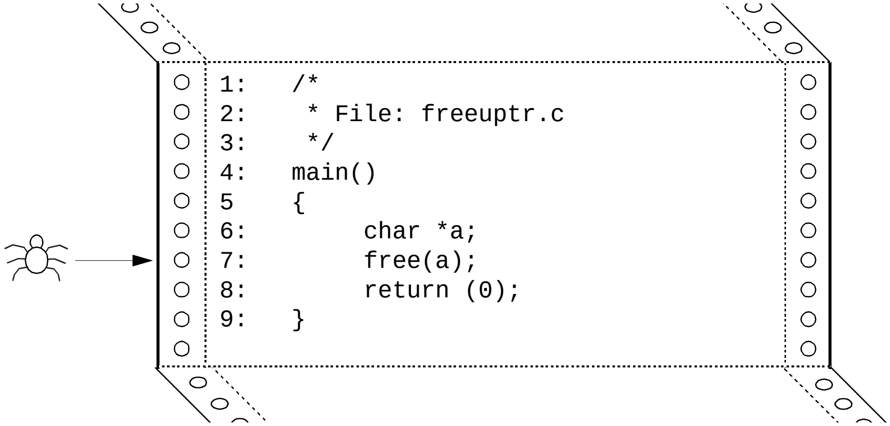
FREE_UNINIT_PTR

Freeing uninitialized pointer

This error is generated whenever an attempt is made to de-allocate memory by means of an uninitialized pointer.

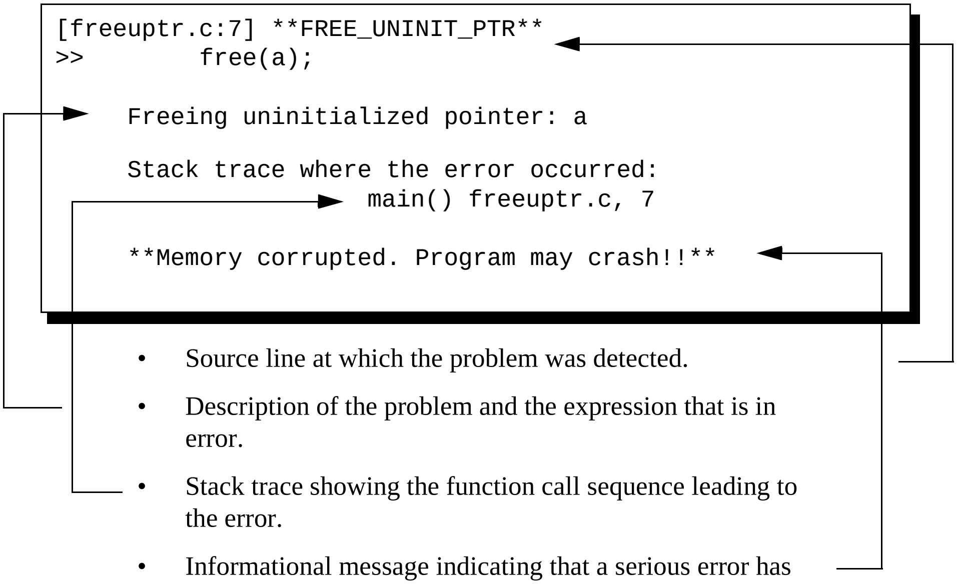
Problem

This code attempts to free a pointer that has not been initialized.



```
/*
 * File: freeuptr.c
 */
main()
{
     char *a;
     free(a);
     return (0);
}
```

Diagnosis (at runtime)



```
[freeuptr.c:7] **FREE_UNINIT_PTR**
>>        free(a);

     Freeing uninitialized pointer: a

     Stack trace where the error occurred:
                      main() freeuptr.c, 7

     **Memory corrupted. Program may crash!!**
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Stack trace showing the function call sequence leading to the error.
- Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

Some systems appear to allow this operation, since they will refuse to free memory that was not dynamically allocated. Relying on this behavior is very dangerous, however, since an uninitialized pointer may “accidentally” point to a block of memory that *was* dynamically allocated, but should not be freed.

FREE_WILD

Freeing wild pointer

This error is generated when memory is de-allocated that is unknown to Insure++.

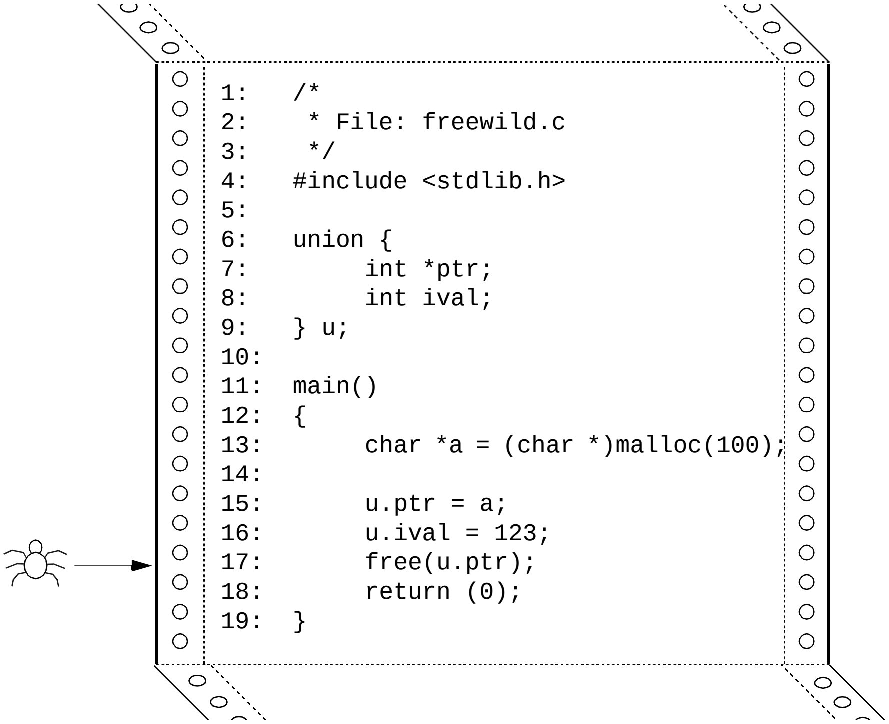
This can come about in two ways:

- Errors in user code that result in pointers that don’t point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

This discussion centers on the first type of problem described here. A detailed discussion of the second topic, including samples of its generation and repair can be found in “Interfaces” on page 103.

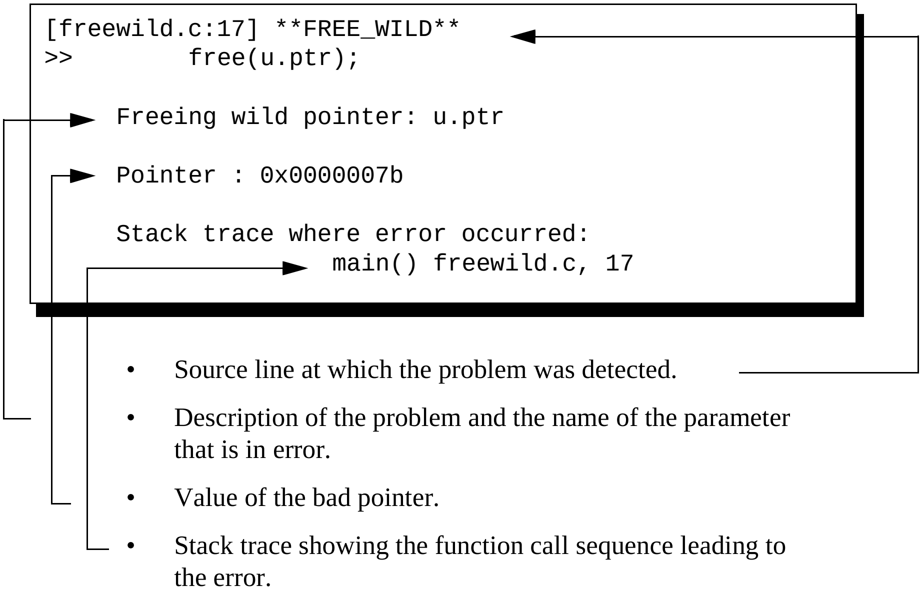
Problem

A particularly unpleasant problem can occur when using `union` types. The following code first assigns the pointer element of a union but then overwrites it with another element before finally attempting to free the initial memory block.



```
/*
 * File: freewild.c
 */
#include <stdlib.h>

union {
     int *ptr;
     int ival;
} u;

main()
{
     char *a = (char *)malloc(100);

     u.ptr = a;
     u.ival = 123;
     free(u.ptr);
     return (0);
}
```

Diagnosis (at runtime)



```
[freewild.c:17] **FREE_WILD**
>>        free(u.ptr);

     Freeing wild pointer: u.ptr

     Pointer : 0x0000007b

     Stack trace where error occurred:
                  main() freewild.c, 17
```

- Source line at which the problem was detected.
- Description of the problem and the name of the parameter that is in error.
- Value of the bad pointer.
- Stack trace showing the function call sequence leading to the error.

Repair

This problem is most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively you can stop the program at some point prior to the error and single-step through the code leading up to the problem.

“Wild pointers” can also be generated when Insure++ has only partial information about your program’s structure. This issue is discussed extensively in “Interfaces” on page 103.

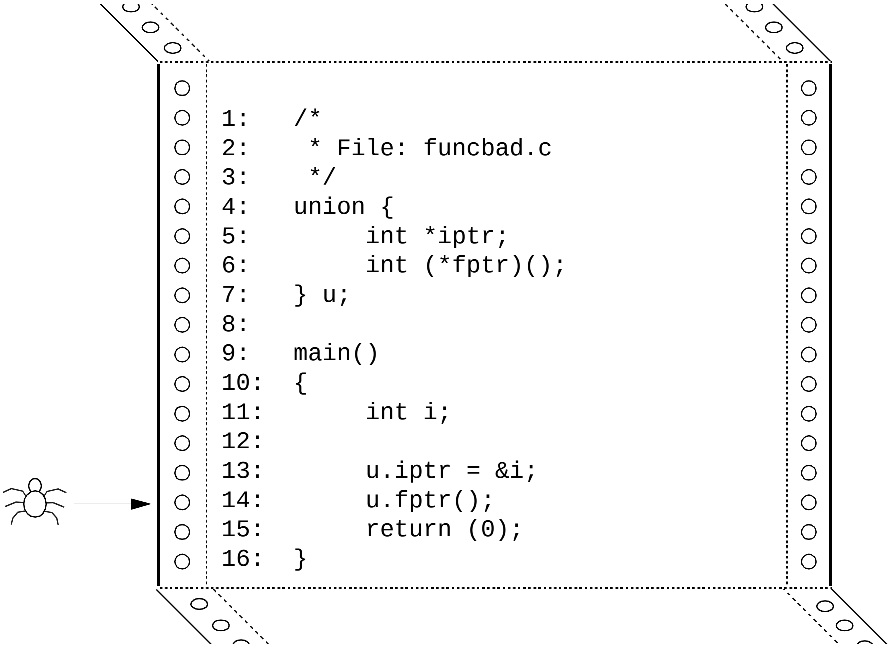
FUNC_BAD

Function pointer is not a function

This error is generated when an attempt is made to call a function through either an invalid or unknown function pointer.

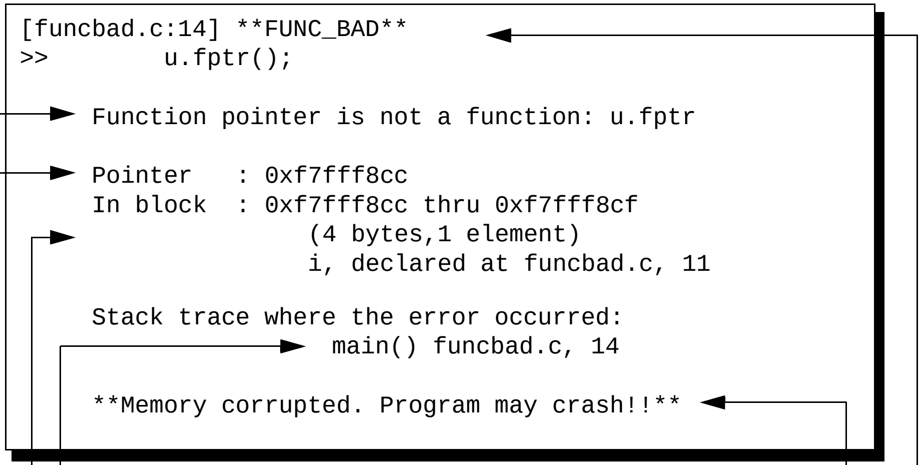
Problem

One simple way to generate this error is through the use of the `union` data type. If the union contains a function pointer which is invoked after initializing some other union member, this error can occur.



```
/*
 * File: funcbad.c
 */
union {
     int *iptr;
     int (*fptr)();
} u;

main()
{
     int i;

     u.iptr = &i;
     u.fptr();
     return (0);
}
```

Diagnosis (at runtime)



```
[funcbad.c:14] **FUNC_BAD**
>>        u.fptr();

     Function pointer is not a function: u.fptr

     Pointer    : 0xf7fff8cc
     In block   : 0xf7fff8cc thru 0xf7fff8cf
                  (4 bytes,1 element)
                  i, declared at funcbad.c, 11

     Stack trace where the error occurred:
                   main() funcbad.c, 14

     **Memory corrupted. Program may crash!!**
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- The value of the pointer through which the call is being attempted.
- Description of the memory block to which this pointer actually points, including its size and the source line of its declaration.
- Stack trace showing the function call sequence leading to the error.
- Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

The description of the memory block to which the pointer points should enable you to identify the statement which was used to assign the function pointer incorrectly.

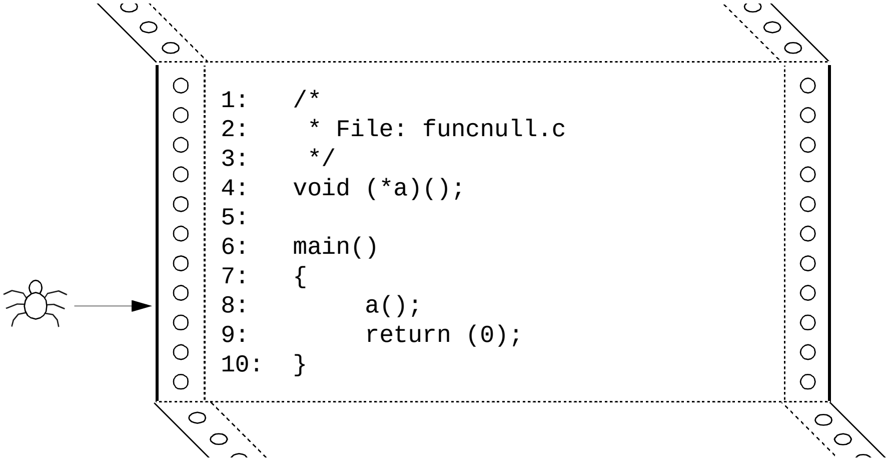
FUNC_NULL

Function pointer is NULL

This error is generated when a function call is made via a NULL function pointer.

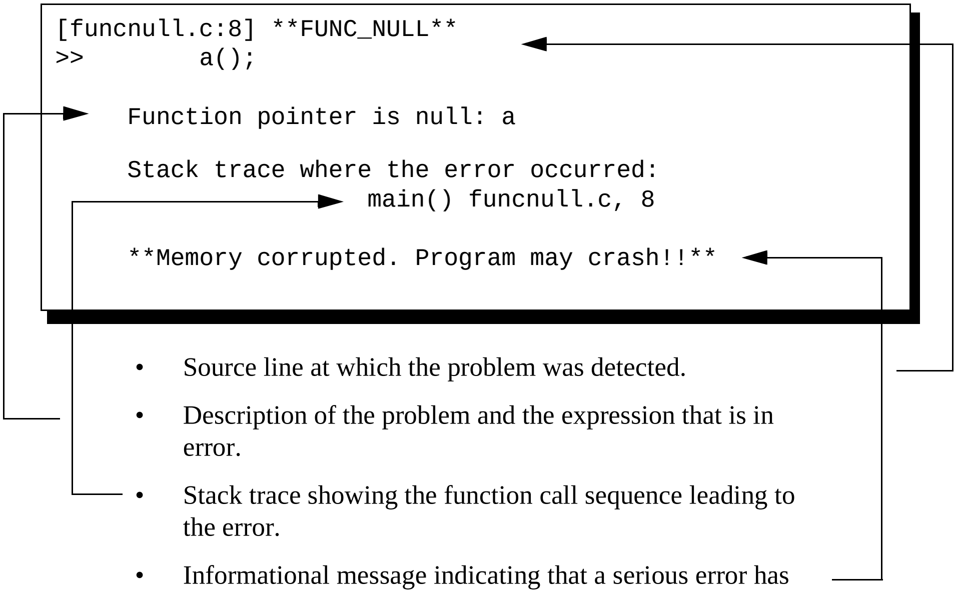
Problem

This code attempts to call a function through a pointer that has never been explicitly initialized. Since the pointer is a global variable, it is initialized to zero by default, resulting in the attempt to call a NULL pointer.



```
/*
 * File: funcnull.c
 */
void (*a)();

main()
{
     a();
     return (0);
}
```

Diagnosis (at runtime)



```
[funcnull.c:8] **FUNC_NULL**
>>        a();

    Function pointer is null: a

    Stack trace where the error occurred:
                    main() funcnull.c, 8

    **Memory corrupted. Program may crash!!**
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Stack trace showing the function call sequence leading to the error.
- Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

The most common way to generate this problem is the one shown here, in which the pointer never got explicitly initialized and is set to zero. This case normally requires the addition of an assignment statement prior to the call as shown below

```
/*
 * File: funcnull.c (modified)
 */
void (*a)();
extern void myfunc();

main()
```

```
{
      a = myfunc;
      a();
      return (0);
}
```

A second fairly common programming practice is to terminate arrays of function pointers with NULL entries. Code that scans a list looking for a particular function may end up calling the NULL pointer if its search criterion fails. This normally indicates that protective programming logic should be added to prevent against this case.

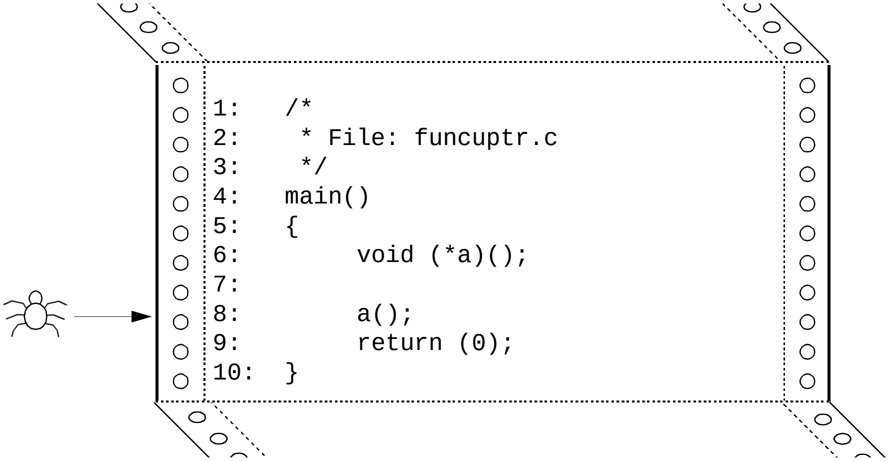
FUNC_UNINIT_PTR

Function pointer is uninitialized

This error is generated when a call is made through an uninitialized function pointer.

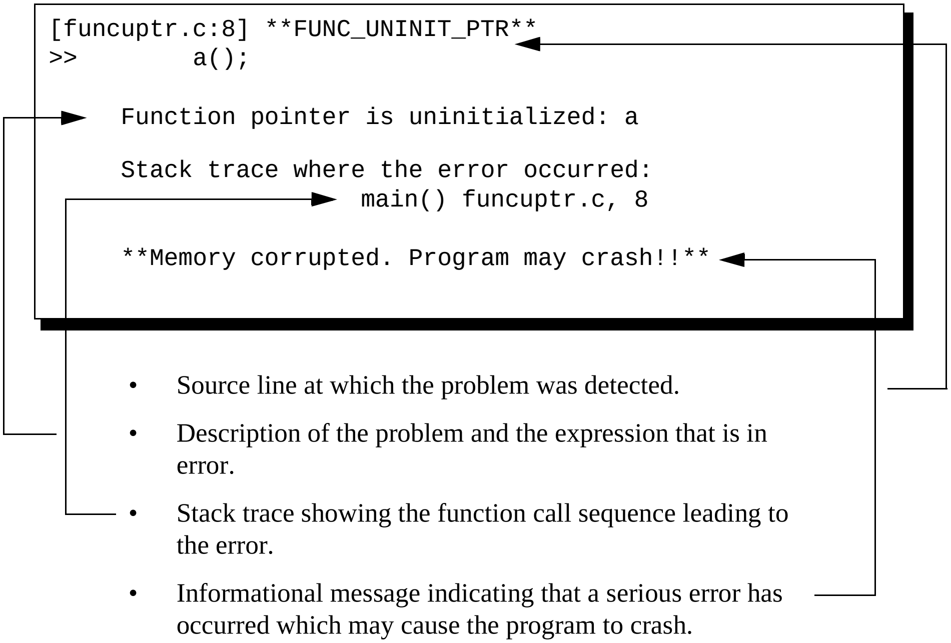
Problem

This code attempts to call a function through a pointer that has not been set.



```
/*
 * File: funcuptr.c
 */
main()
{
     void (*a)();

     a();
     return (0);
}
```

Diagnosis (at runtime)



```
[funcuptr.c:8] **FUNC_UNINIT_PTR**
>>        a();

     Function pointer is uninitialized: a

     Stack trace where the error occurred:
                      main() funcuptr.c, 8

     **Memory corrupted. Program may crash!!**
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Stack trace showing the function call sequence leading to the error.
- Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

This problem normally occurs because some assignment statement has been omitted from the code. The current example can be fixed as follows

```
extern void myfunc();

main()
{
      void (*a)();
      a = myfunc;
      a();
}
```

FUNC_WILD

Function pointer is wild

This error is generated when a call is made via a function pointer that is unknown to Insure++.

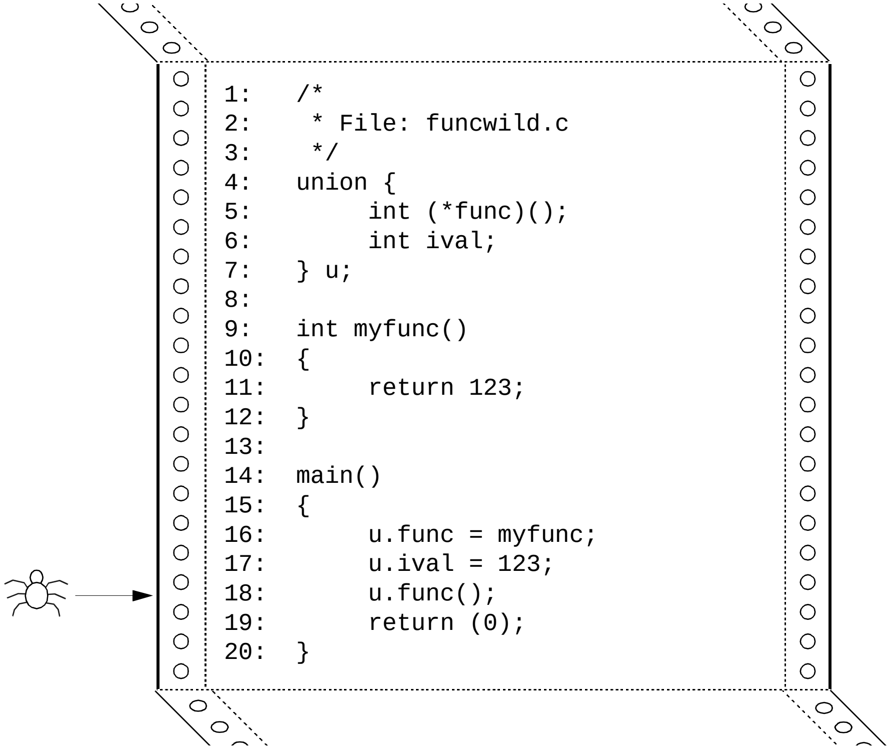
This can come about in two ways:

- Errors in user code that result in pointers that don’t point at any known function.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

This discussion centers on the first type of problem described here. A detailed discussion of the second topic, including samples of its generation and repair can be found in “Interfaces” on page 103.

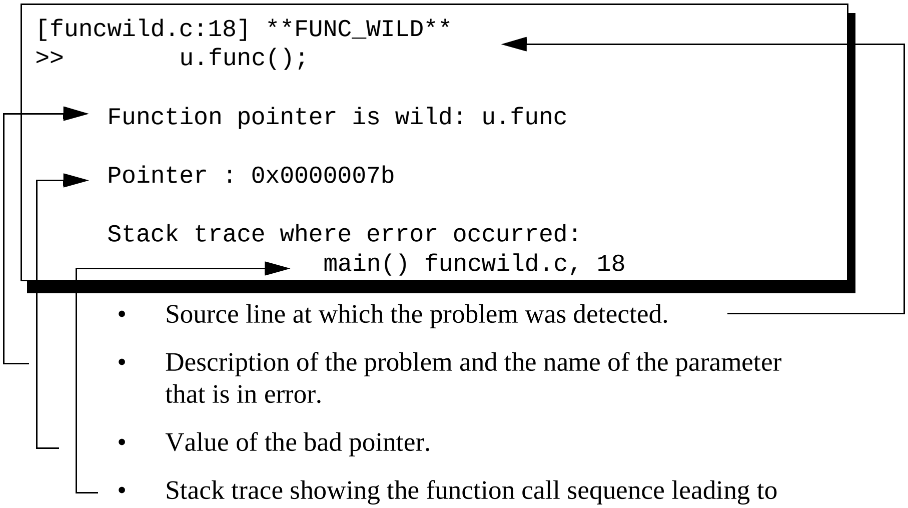
Problem

A particularly unpleasant problem can occur when using `union` types. The following code first assigns the pointer element of a union but then overwrites it with another element before finally attempting the function call at line 18.



```
1:   /*
2:    * File: funcwild.c
3:    */
4:   union {
5:        int (*func)();
6:        int ival;
7:   } u;
8:
9:   int myfunc()
10:  {
11:       return 123;
12:  }
13:
14:  main()
15:  {
16:       u.func = myfunc;
17:       u.ival = 123;
18:       u.func();
19:       return (0);
20:  }
```

Diagnosis (at runtime)



```
[funcwild.c:18] **FUNC_WILD**
>>        u.func();

     Function pointer is wild: u.func

     Pointer : 0x0000007b

     Stack trace where error occurred:
                    main() funcwild.c, 18
```

- Source line at which the problem was detected.
- Description of the problem and the name of the parameter that is in error.
- Value of the bad pointer.
- Stack trace showing the function call sequence leading to the error.

Repair

This problem is most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively you can stop the program at some point prior to the error and single-step through the code leading up to the problem.

Note that wild pointers can also be generated when Insure++ has only partial information about your program’s structure. This issue is discussed extensively in “Interfaces” on page 103.

INSURE_ERROR

Internal errors (various)

This error code is reserved for fatal errors that Insure++ is unable to deal with adequately such as running out of memory, or failing to open a required file.

Unrecognized options in `.psrc` files can also generate this error.

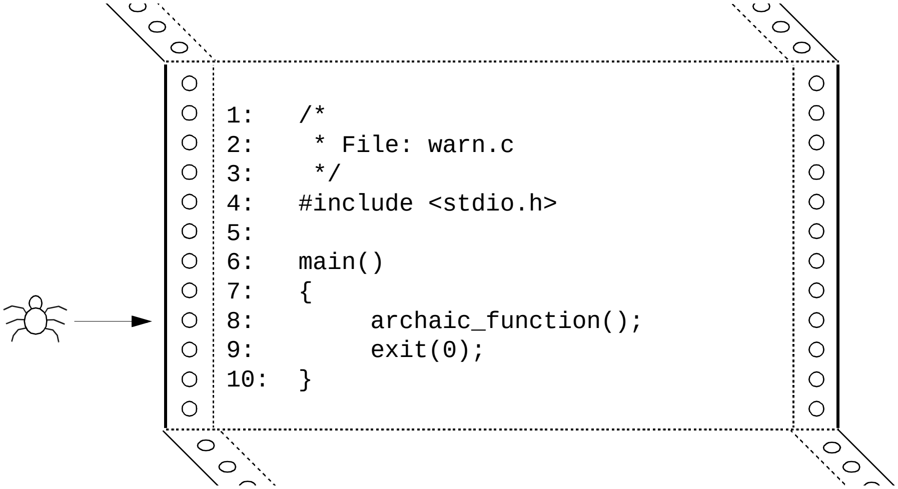
INSURE++_WARNING

Errors from `iic_warning` calls

This error code is generated when Insure++ encounters a call to the `iic_warning` interface function.

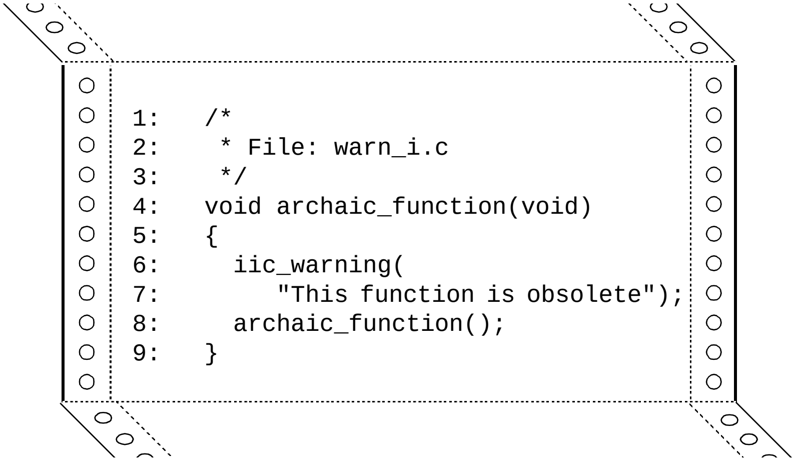
Example

The following code contains a call to a function called `archaic_function` whose use is to be discouraged.



```
/*
 * File: warn.c
 */
#include <stdio.h>

main()
{
        archaic_function();
        exit(0);
}
```

In order to use the `iic_warning` capability, we can make an interface to the `archaic_function` as follows.



```
/*
 * File: warn_i.c
 */
void archaic_function(void)
{
  iic_warning(
     "This function is obsolete");
  archaic_function();
}
```

Diagnosis (during compilation)

```
[warn.c:8] **INSURE++_WARNING**
     Use of archaic_function is deprecated.
>>        archaic_function();
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.

Repair

This error category is suppressed by default, so you must add the option

```
insure++.unsuppress INSURE_WARNING
```

to your .psrc file before compiling code which uses it.

There are many uses for iic_warning and the INSURE_WARNING error, so no specific suggestions for error correction are appropriate. Hopefully, the messages displayed by the system will provide sufficient assistance.

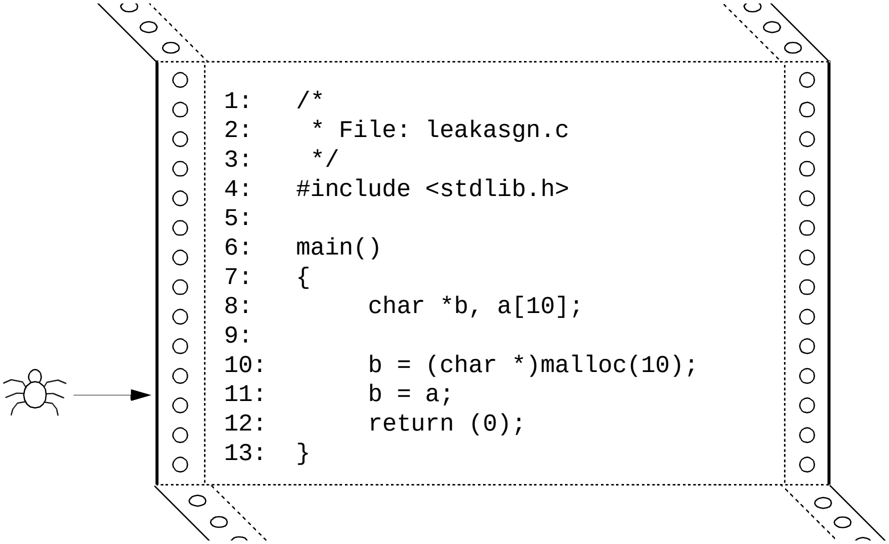
LEAK_ASSIGN

Memory leaked due to pointer reassignment

This error is generated whenever a pointer assignment occurs which will prevent a block of dynamically allocated memory from ever being freed. Normally this happens because the pointer being changed is the only one that still points to the dynamically allocated block.

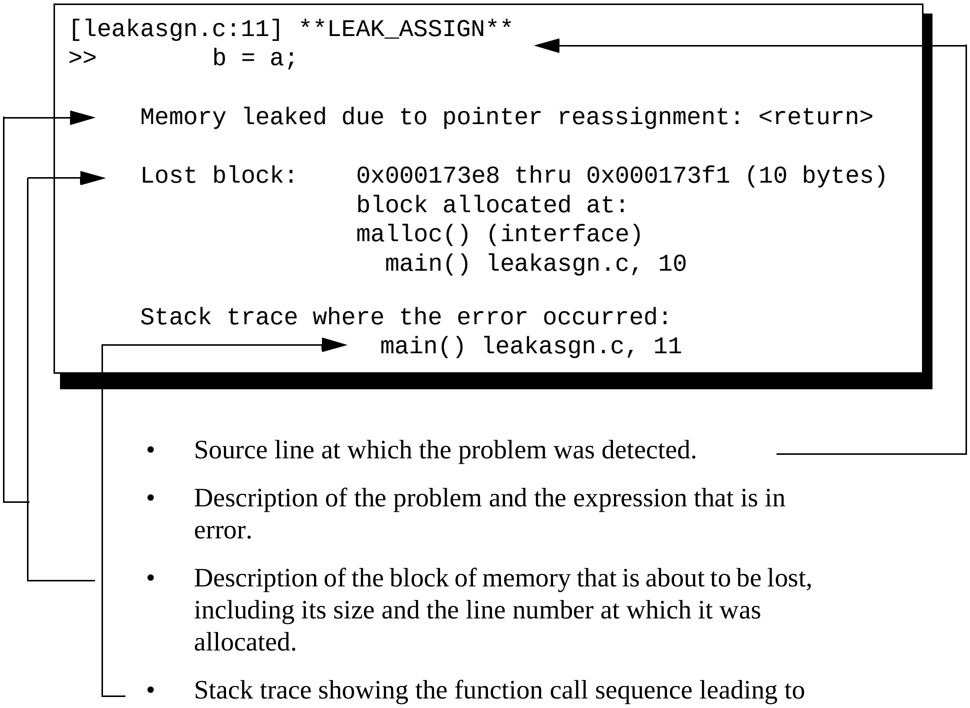
Problem

This code allocates a block of memory, but then reassigns the pointer to the block to a static memory block. As a result, the dynamically allocated block can no longer be freed.



```
/*
 * File: leakasgn.c
 */
#include <stdlib.h>

main()
{
     char *b, a[10];

     b = (char *)malloc(10);
     b = a;
     return (0);
}
```

Diagnosis (at runtime)



```
[leakasgn.c:11] **LEAK_ASSIGN**
>>        b = a;

     Memory leaked due to pointer reassignment: <return>

     Lost block:    0x000173e8 thru 0x000173f1 (10 bytes)
                    block allocated at:
                    malloc() (interface)
                      main() leakasgn.c, 10

     Stack trace where the error occurred:
                      main() leakasgn.c, 11
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Description of the block of memory that is about to be lost, including its size and the line number at which it was allocated.
- Stack trace showing the function call sequence leading to the error.

Repair

In many cases, this problem is caused by simply forgetting to free a previously allocated block of memory when a pointer is reassigned. For example, the leak in the example code can be corrected as follows

```
10:   b = (char *)malloc(10);
11:   free(b);
12:   b = a;
```

Some applications may be unable to free memory blocks and may not need to worry about their permanent loss. To suppress these error messages, add the option

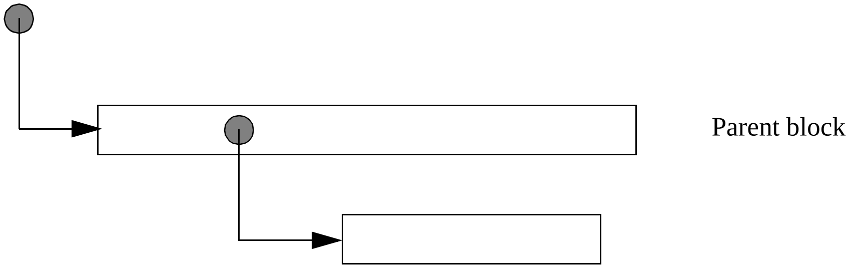
```
insure++.suppress LEAK_ASSIGN
```

to your `.psrc` file.

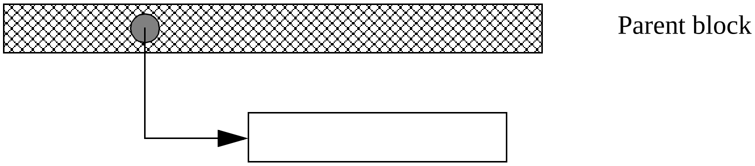
LEAK_FREE

Memory leaked freeing block

This problem can occur when a block of memory contains a pointer to another dynamically allocated block, as indicated in the following figure.

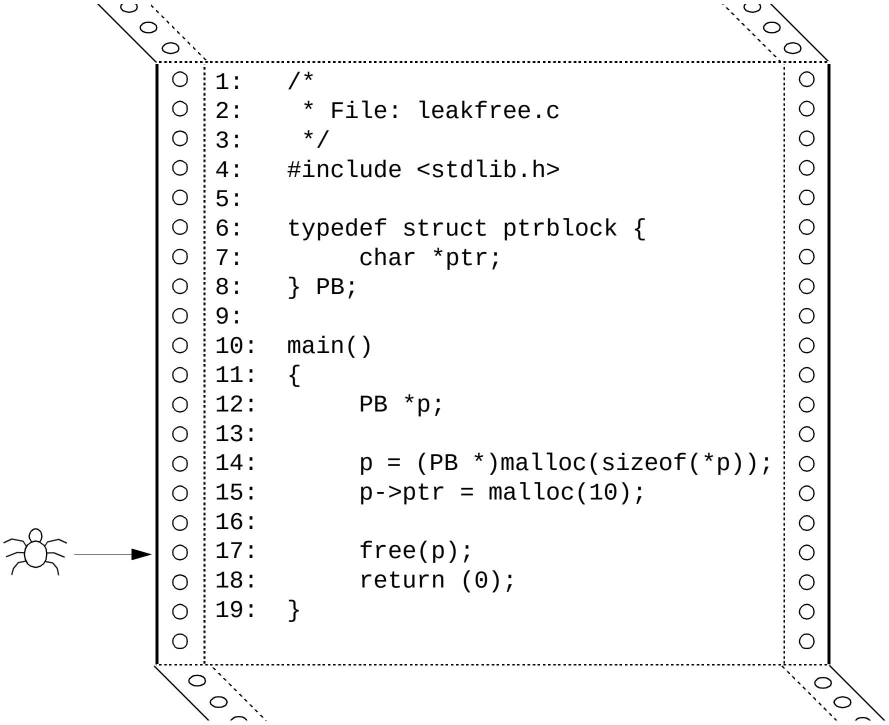


If the main memory block is freed its memory becomes invalid, which means that the included pointer can no longer be used to free the second block. This causes a permanent memory leak.



Problem

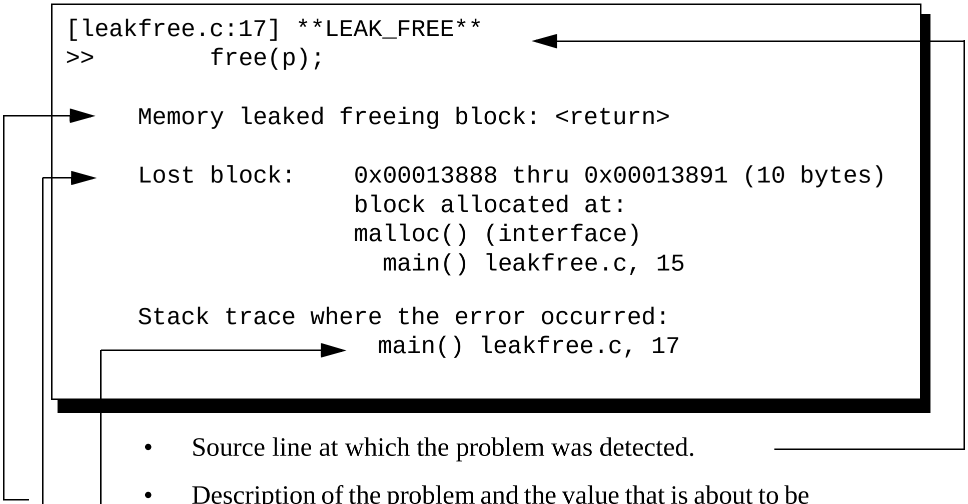
This code defines PB to be a data structure that contains a pointer to another block of memory.



```
/*
 * File: leakfree.c
 */
#include <stdlib.h>

typedef struct ptrblock {
     char *ptr;
} PB;

main()
{
     PB *p;

     p = (PB *)malloc(sizeof(*p));
     p->ptr = malloc(10);

     free(p);
     return (0);
}
```

We first create a single PB and then allocate a block of memory for it to point to. The call to free on the PB then causes a permanent memory leak, since it frees the memory containing the only pointer to the second allocated block. This latter block can no longer be freed.

Diagnosis (at runtime)



```
[leakfree.c:17] **LEAK_FREE**
>>        free(p);

     Memory leaked freeing block: <return>

     Lost block:    0x00013888 thru 0x00013891 (10 bytes)
                    block allocated at:
                    malloc() (interface)
                      main() leakfree.c, 15

     Stack trace where the error occurred:
                      main() leakfree.c, 17
```

- Source line at which the problem was detected.
- Description of the problem and the value that is about to be lost.
- Description of the block of memory that is about to be lost, including its size and the line number at which it was allocated.
- Stack trace showing the function call sequence leading to the error.

Repair

In many cases, this problem is caused by forgetting to free the enclosed blocks when freeing their container. This can be corrected by adding a suitable call to free the memory before freeing the parent block.

Caution must be used when doing this, however, to ensure that the memory blocks are freed in the correct order. Changing the example in the following manner, for example, would still generate the same error:

```
free(p);
free(p->ptr);
```

because the blocks are freed in the wrong order. The contained blocks must be freed before their parents, because the memory becomes invalid as soon as it is freed. Thus, the second call to `free` in the above code fragment might fail, because the value `p->ptr` is no longer valid. It is quite legal, for example, for the first call to `free` to have set to zero or otherwise destroyed the contents of its memory block.[1]

Some applications may be unable to free memory blocks and may not need to worry about their permanent loss. To suppress these error messages in this case add the option

```
insure++.suppress LEAK_FREE
```

to your `.psrc` file.

[1] Many systems allow the out of order behavior, although it is becoming less portable as more and more systems move to dynamically re-allocated (moveable) memory blocks.

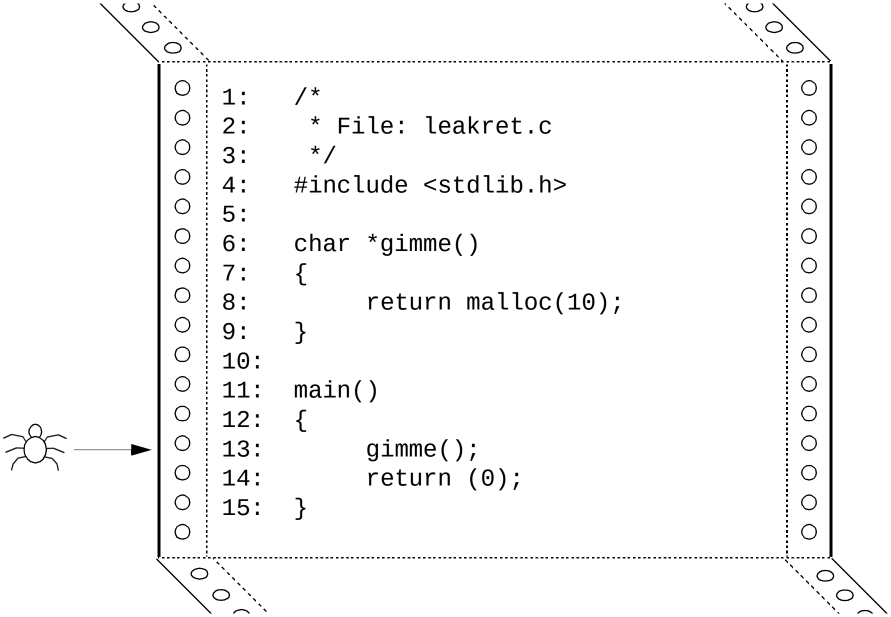
LEAK_RETURN

Memory leaked by ignoring returned value

This error is generated whenever a function returns a pointer to a block of memory which is then ignored by the calling routine. In this case, the allocated memory block is permanently lost and can never be freed.

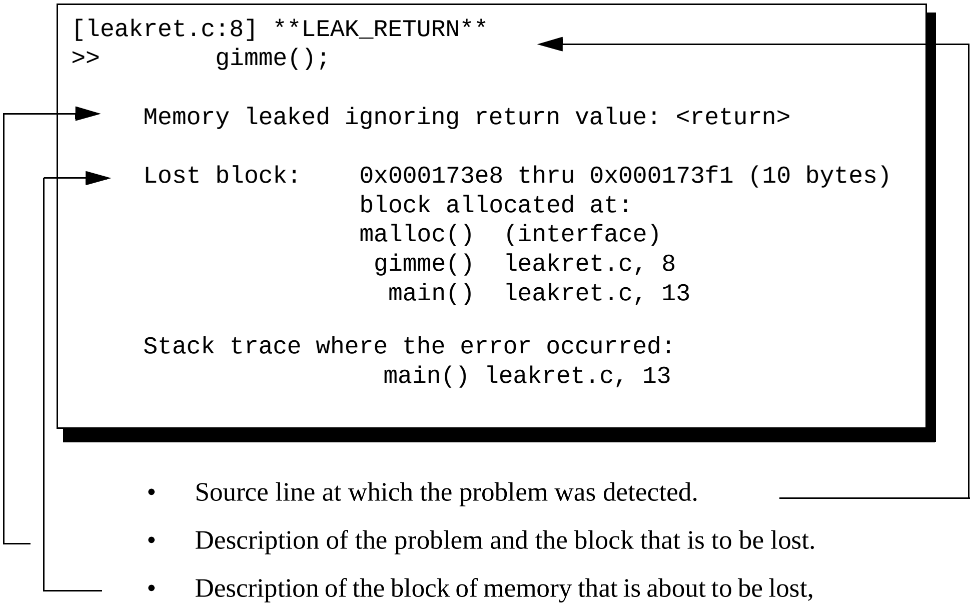
Problem

This code calls the function `gimme`, which returns a memory block that is subsequently ignored by the `main` routine.



```
/*
 * File: leakret.c
 */
#include <stdlib.h>

char *gimme()
{
     return malloc(10);
}

main()
{
     gimme();
     return (0);
}
```

Diagnosis (at runtime)



```
[leakret.c:8] **LEAK_RETURN**
>>        gimme();

     Memory leaked ignoring return value: <return>

     Lost block:    0x000173e8 thru 0x000173f1 (10 bytes)
                    block allocated at:
                    malloc()  (interface)
                     gimme()  leakret.c, 8
                      main()  leakret.c, 13

     Stack trace where the error occurred:
                      main() leakret.c, 13
```

- Source line at which the problem was detected.
- Description of the problem and the block that is to be lost.
- Description of the block of memory that is about to be lost, including its size and the line number at which it was allocated.

Repair

This problem usually results from an oversight on the part of the programmer, or a misunderstanding of the nature of the pointer returned by a routine. In particular, it is sometimes unclear whether the value returned points to a static block of memory, which will not need to be freed, or a dynamically allocated one, which should be.

Some applications may be unable to free memory blocks and may not need to worry about their permanent loss. To suppress these error messages in this case add the option

```
insure++.suppress LEAK_RETURN
```

to your `.psrc` file.

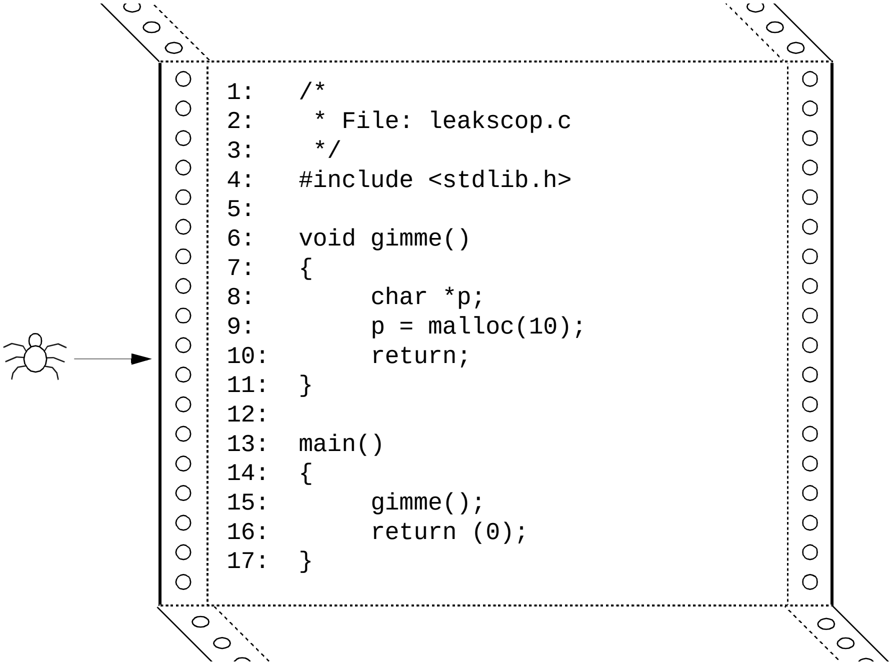
LEAK_SCOPE

Memory leaked leaving scope

This error is generated whenever a function allocates memory for its own use and then returns without freeing it or saving a pointer to the block in an external variable. The allocated block can never be freed.

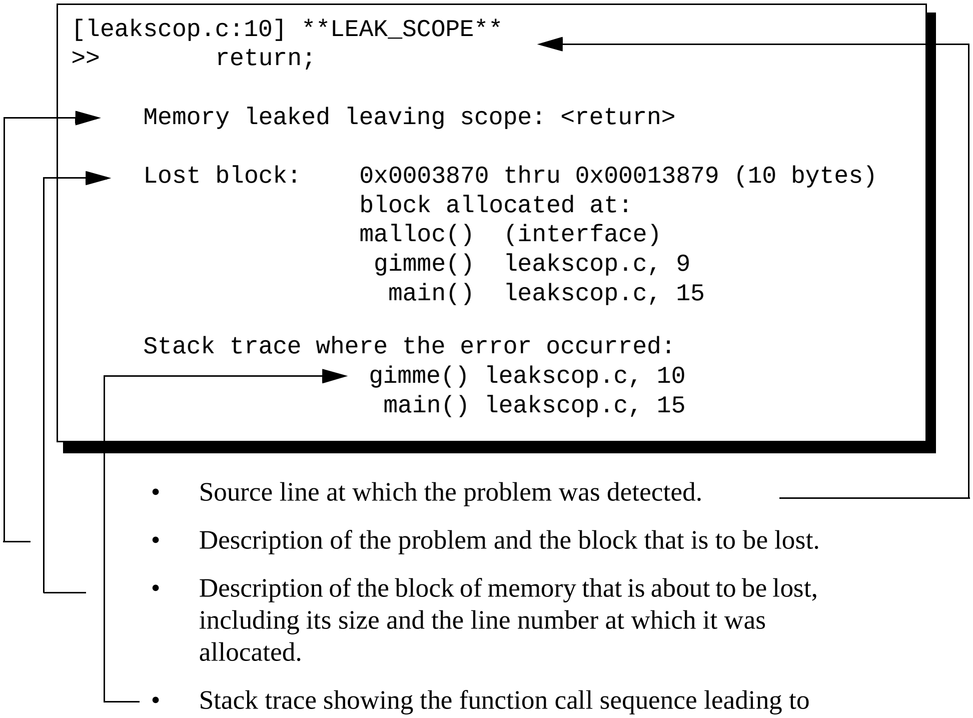
Problem

This code calls the function `gimme`, which allocates a memory block that is never freed.



```
1:   /*
2:    * File: leakscop.c
3:    */
4:   #include <stdlib.h>
5:
6:   void gimme()
7:   {
8:        char *p;
9:        p = malloc(10);
10:       return;
11:  }
12:
13:  main()
14:  {
15:       gimme();
16:       return (0);
17:  }
```

Diagnosis (at runtime)



```
[leakscop.c:10] **LEAK_SCOPE**
>>        return;

    Memory leaked leaving scope: <return>

    Lost block:     0x0003870 thru 0x00013879 (10 bytes)
                    block allocated at:
                    malloc()  (interface)
                     gimme()  leakscop.c, 9
                      main()  leakscop.c, 15

    Stack trace where the error occurred:
                     gimme() leakscop.c, 10
                      main() leakscop.c, 15
```

- Source line at which the problem was detected.
- Description of the problem and the block that is to be lost.
- Description of the block of memory that is about to be lost, including its size and the line number at which it was allocated.
- Stack trace showing the function call sequence leading to the error.

Repair

This problem usually results from an oversight on the part of the programmer and is cured by simply freeing a block before returning from a routine. In the current example, a call to `free(p)` before line 10 would cure the problem.

A particularly easy way to generate this error is to return from the middle of a routine, possibly due to an error condition arising, without freeing previously allocated data. This bug is easy to introduce when modifying existing code.

Some applications may be unable to free memory blocks and may not need to worry about their permanent loss. To suppress these error messages in this case add the option

```
insure++.suppress LEAK_SCOPE
```

to your `.psrc` file.

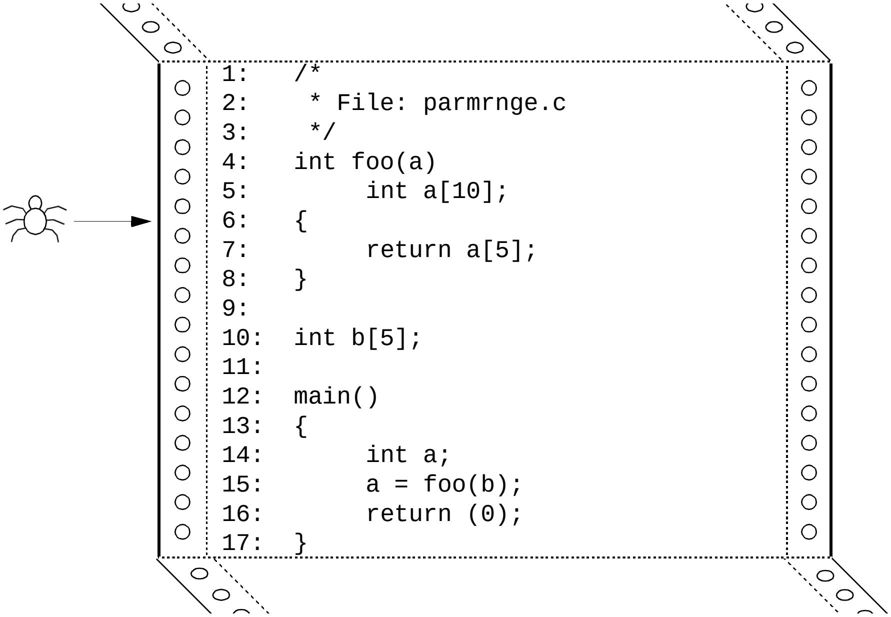
PARM_BAD_RANGE

Array parameter exceeded range

This error is generated whenever a function parameter is declared as an array, but has more elements than the actual argument which was passed.

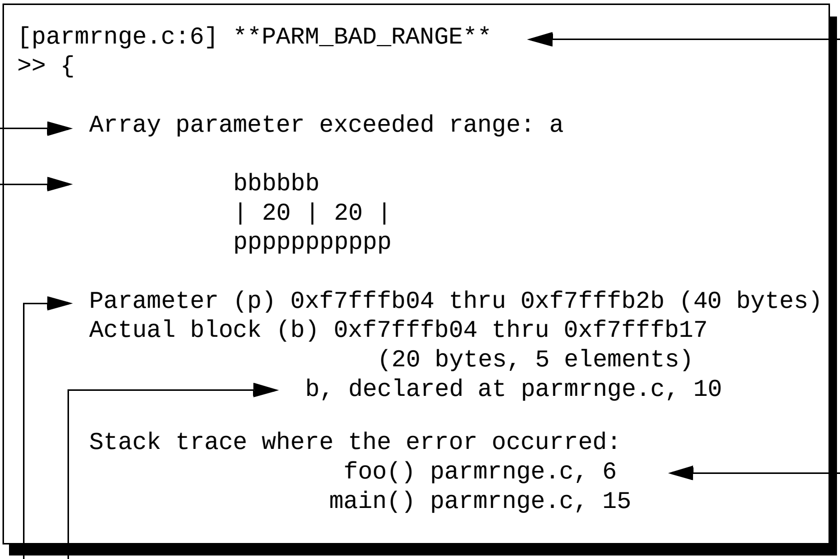
Problem

The following code fragment shows an array declared with one size in the `main` routine and then used with another in a function.



```
/*
 * File: parmrnge.c
 */
int foo(a)
     int a[10];
{
     return a[5];
}

int b[5];

main()
{
     int a;
     a = foo(b);
     return (0);
}
```

Diagnosis (at runtime)



```
[parmrnge.c:6] **PARM_BAD_RANGE**
>> {

     Array parameter exceeded range: a

               bbbbbb
               | 20 | 20 |
               ppppppppppp

     Parameter (p) 0xf7fffb04 thru 0xf7fffb2b (40 bytes)
     Actual block (b) 0xf7fffb04 thru 0xf7fffb17
                          (20 bytes, 5 elements)
                    b, declared at parmrnge.c, 10

     Stack trace where the error occurred:
                          foo() parmrnge.c, 6
                         main() parmrnge.c, 15
```

- Source line at which the problem was detected.
- Description of the problem and the name of the parameter that is in error.
- Schematic showing the relative layout of the memory block which was actually passed as the argument (b) and expected parameter (p). (See “Overflow diagrams” on page 169.)
- Description of the memory range occupied by the parameter, including its length.
- Description of the actual block of data corresponding to the argument, including its address range and size. Also includes the name of the real variable which matches the argument and the line number at which it was declared.
- Stack trace showing the function call sequence leading to the error.

Repair

This error is normally straightforward to correct based on the information presented in the diagnostic output.

The simplest solution is to change the definition of the array in the called routine to indicate an array of unknown size, i.e., replace line 5 with

```
parmrnge.c, 5  int a[];
```

This declaration will match any array argument and is the recommended approach whenever the called routine will accept arrays of variable size.

An alternative is to change the declaration of the array in the calling routine to match that expected. In this case, line 10 could be changed to

```
parmrnge.c, 10 int b[10];
```

which now matches the argument declaration.

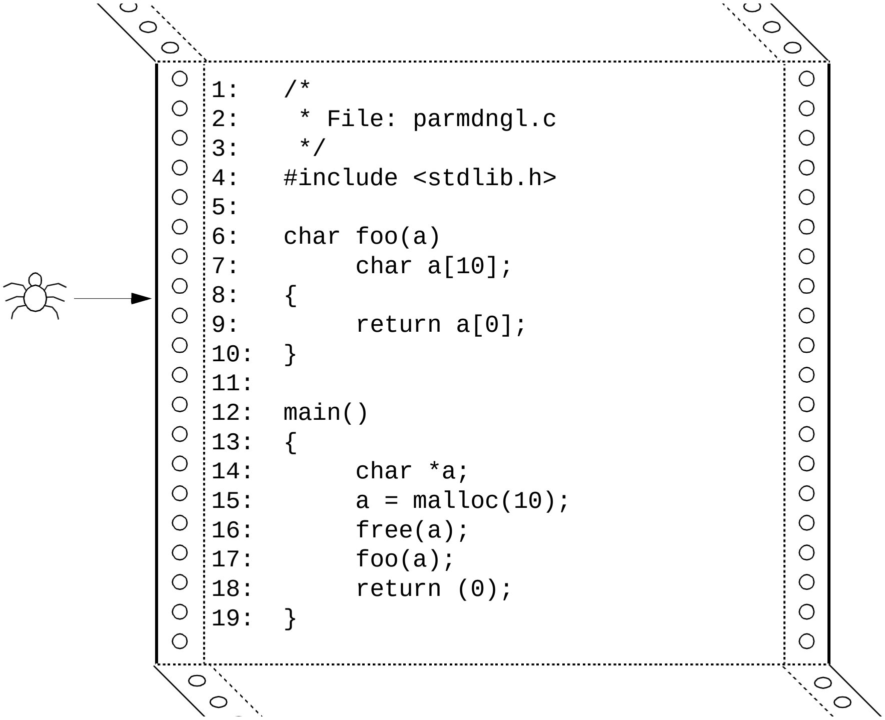
PARM_DANGLING

Array parameter is dangling pointer

This error is generated whenever a parameter declared as an array is actually passed a pointer to a block of memory that has been freed.

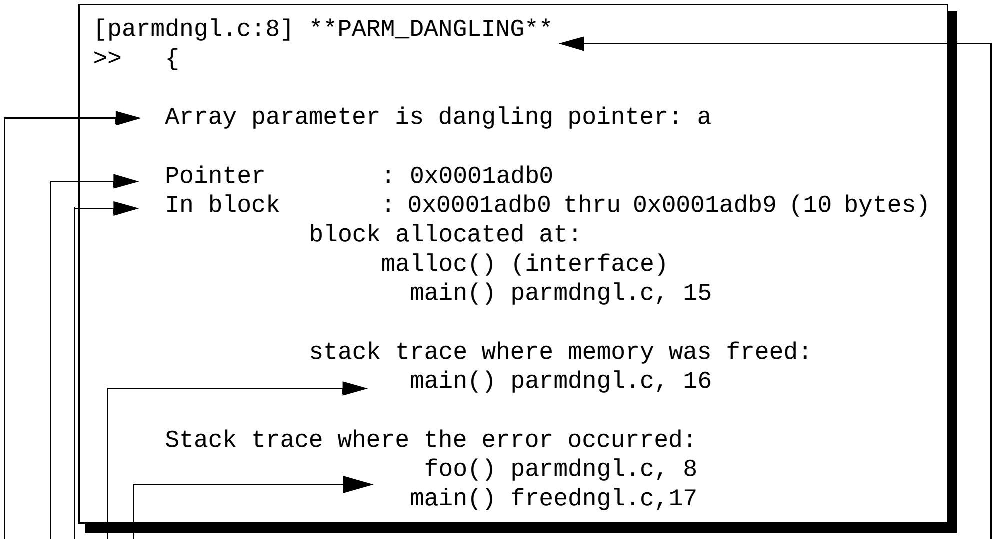
Problem

The following code frees its memory block before passing it to `foo`.



```
/*
 * File: parmdngl.c
 */
#include <stdlib.h>

char foo(a)
     char a[10];
{
     return a[0];
}

main()
{
     char *a;
     a = malloc(10);
     free(a);
     foo(a);
     return (0);
}
```

Diagnosis (at runtime)



```
[parmdngl.c:8] **PARM_DANGLING**
>>   {

     Array parameter is dangling pointer: a

     Pointer        : 0x0001adb0
     In block       : 0x0001adb0 thru 0x0001adb9 (10 bytes)
               block allocated at:
                    malloc() (interface)
                      main() parmdngl.c, 15

               stack trace where memory was freed:
                      main() parmdngl.c, 16

     Stack trace where the error occurred:
                       foo() parmdngl.c, 8
                      main() freedngl.c,17
```

- Source line at which the problem was detected.
- Description of the problem and the parameter that is in error.
- Value of the pointer that was passed and has been deallocated.
- Information about the block of memory addressed by this pointer, including information about where this block was allocated.
- Indication of the line at which this block was freed.
- Stack trace showing the function call sequence leading to the error.

Repair

This error is normally caused by freeing a piece of memory too soon.

A good strategy is to examine the line of code indicated by the diagnostic message which shows where the memory block was freed and check that it should indeed have been de-allocated.

A second check is to verify that the correct parameter was passed to the subroutine.

A third strategy which is sometimes useful is to NULL pointers that have been freed and then check in the called subroutine for this case. Code similar to the following is often useful

```
#include <stdlib.h>

char foo(a)
     char *a;
{
     if(a) return a[0];
     return '!';
}

main()
{
     char *a;
     a = (char *)malloc(10);
     free(a);
     a = NULL;
     foo(a);
     return (0);
}
```

The combination of resetting the pointer to NULL after freeing it and the check in the called subroutine prevents misuse of dangling pointers.

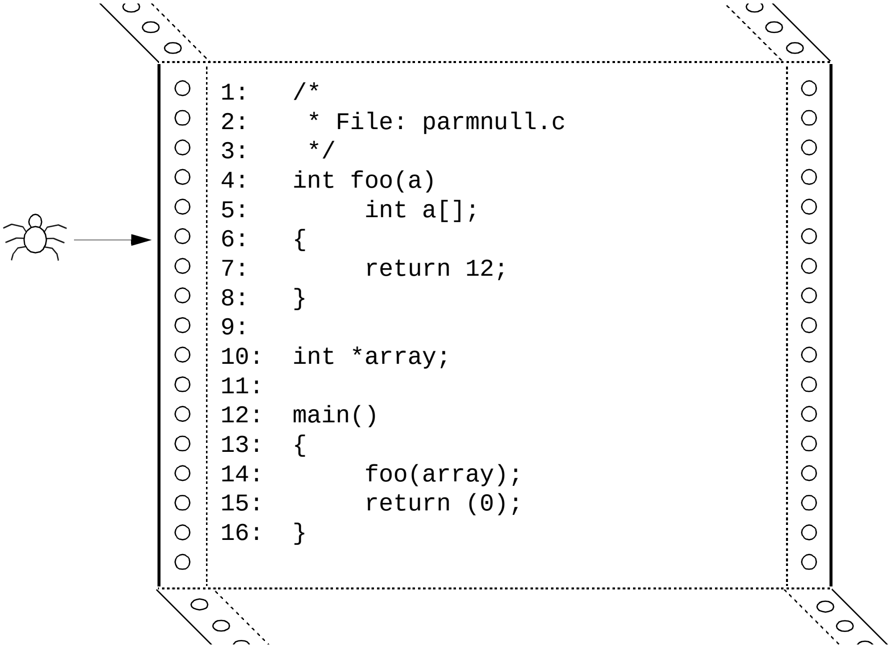
PARM_NULL

Array parameter is NULL

This error is generated whenever a parameter declared as an array is actually passed a NULL pointer.

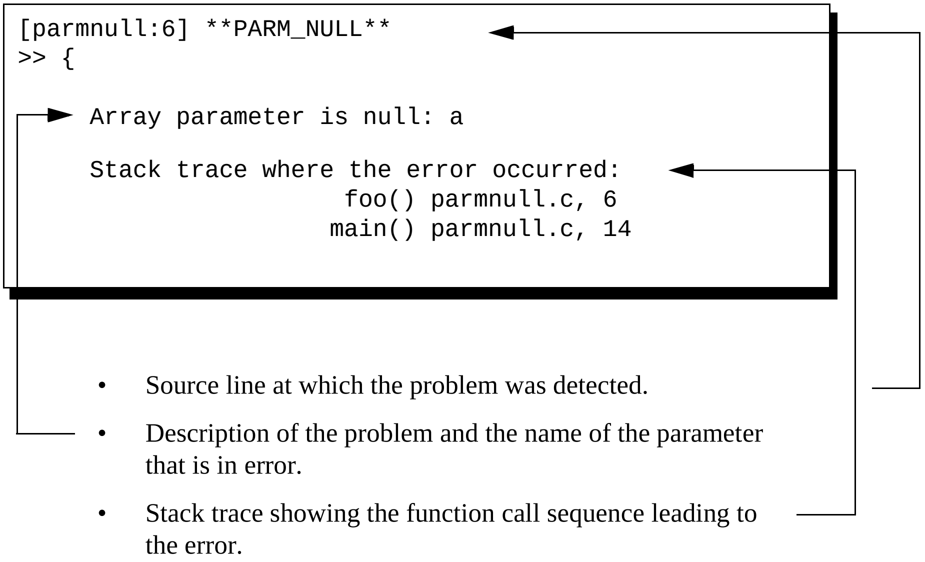
Problem

The following code fragment shows a function which is declared as having an array parameter, but which is invoked with a NULL pointer. The value of array is NULL because it is a global variable, initialized to zero by default.



```
/*
 * File: parmnull.c
 */
int foo(a)
     int a[];
{
     return 12;
}

int *array;

main()
{
     foo(array);
     return (0);
}
```

Diagnosis (at runtime)



```
[parmnull:6] **PARM_NULL**
>> {

     Array parameter is null: a

     Stack trace where the error occurred:
                       foo() parmnull.c, 6
                      main() parmnull.c, 14
```

- Source line at which the problem was detected.
- Description of the problem and the name of the parameter that is in error.
- Stack trace showing the function call sequence leading to the error.

Repair

A common cause of this error is the one given in this example, a global pointer which is initialized to zero by the compiler and then never reassigned. The correction for this case is to include code to initialize the pointer, possibly by allocating dynamic memory or by assigning it to some other array object.

For example, we could change the `main` routine of the example to

```
main()
{
      int local[10];

      array = local;
      foo(array);
}
```

This problem can also occur when a pointer is set to NULL by the code (perhaps to indicate a freed block of memory) and then passed to a routine that expects an array as an argument.

In this case, Insure++ distinguishes between functions whose arguments are declared as arrays

```
int foo(int a[])
{
```

and those with pointer arguments

```
int foo(int *a)
{
```

The latter type will not generate an error if passed a NULL argument, while the former will.

A final common problem is caused when one of the dynamic memory allocation routines, malloc, calloc, or realloc, fails and returns a NULL pointer. This can happen either because your program passes bad arguments or simply because it asks for too much memory. A simple way of finding this problem with Insure++ is to enable the RETURN_FAILURE error code (see page 342) via your .psrc file and run the program again. It will then issue diagnostic messages every time a system call fails, including the memory allocation routines.

If your application cannot avoid passing a NULL pointer to a routine, you should either change the declaration of its argument to the second style or suppress these error messages by adding the option

```
insure++.suppress PARM_NULL
```

to your .psrc file.

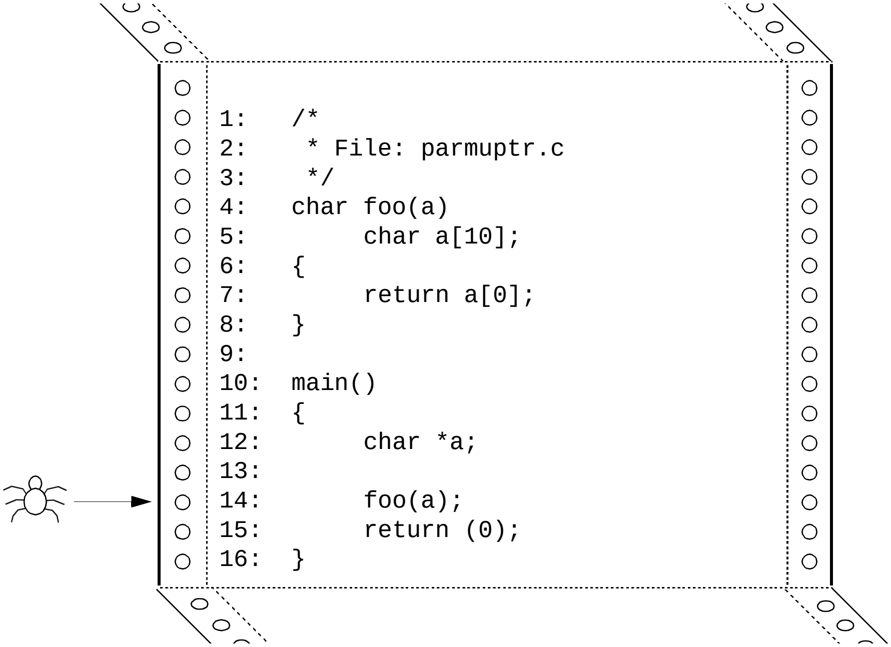
PARM_UNINIT_PTR

Array parameter is uninitialized pointer

This error is generated whenever an uninitialized pointer is passed as an argument to a function which expects an array parameter.

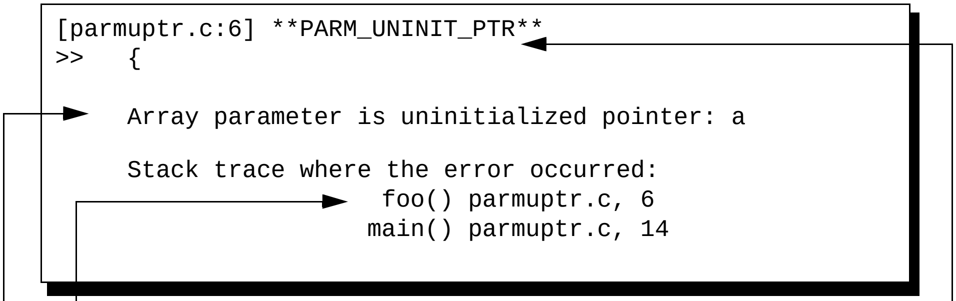
Problem

This code passes the uninitialized pointer `a` to routine `foo`.



```
/*
 * File: parmuptr.c
 */
char foo(a)
     char a[10];
{
     return a[0];
}

main()
{
     char *a;

     foo(a);
     return (0);
}
```

Diagnosis (at runtime)



```
[parmuptr.c:6] **PARM_UNINIT_PTR**
>>   {

     Array parameter is uninitialized pointer: a

     Stack trace where the error occurred:
                       foo() parmuptr.c, 6
                      main() parmuptr.c, 14
```

- Source line at which the problem was detected.
- Description of the problem and the argument that is in error.
- Stack trace showing the function call sequence leading to the error

Repair

This problem is usually caused by omitting an assignment or allocation statement that would initialize a pointer. The code given, for example, could be corrected by including an assignment as shown below.

```
/*
 * File: parmuptr.c (Modified)
 */
...
main()
{
      char *a, b[10];
      a = b;
      foo(a);
}
```

PARM_WILD

Array parameter is wild

This error is generated whenever a parameter is declared as an array but the actual value passed when the function is called points to no known memory block.

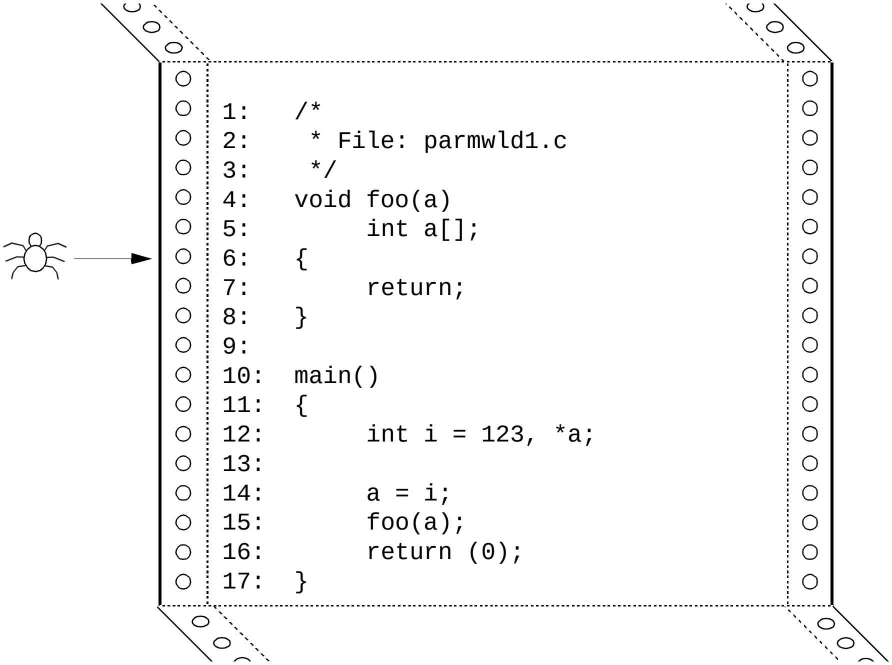
This can come about in several ways:

- Errors in user code that result in pointers that don't point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

This discussion centers on the first type of problem described here. A detailed discussion of the second topic, including samples of its generation and repair can be found in "Interfaces" on page 103.

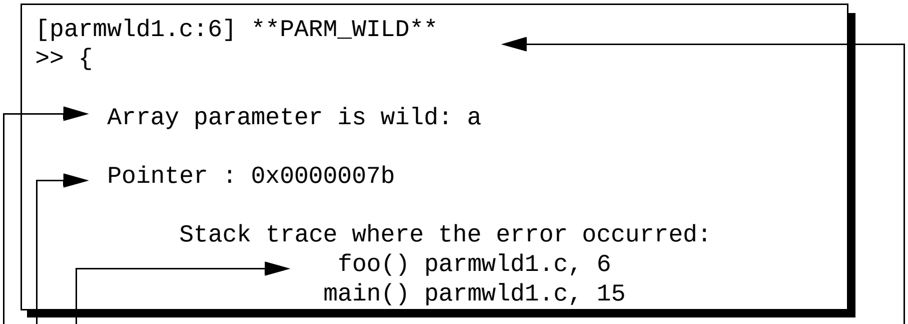
Problem #1

The following code attempts to pass the address of a local variable to the routine `foo` but contains an error at line 14 - the address operator (&) has been omitted.



```
1:   /*
2:    * File: parmwld1.c
3:    */
4:   void foo(a)
5:        int a[];
6:   {
7:        return;
8:   }
9:
10:  main()
11:  {
12:       int i = 123, *a;
13:
14:       a = i;
15:       foo(a);
16:       return (0);
17:  }
```

Diagnosis (at runtime)



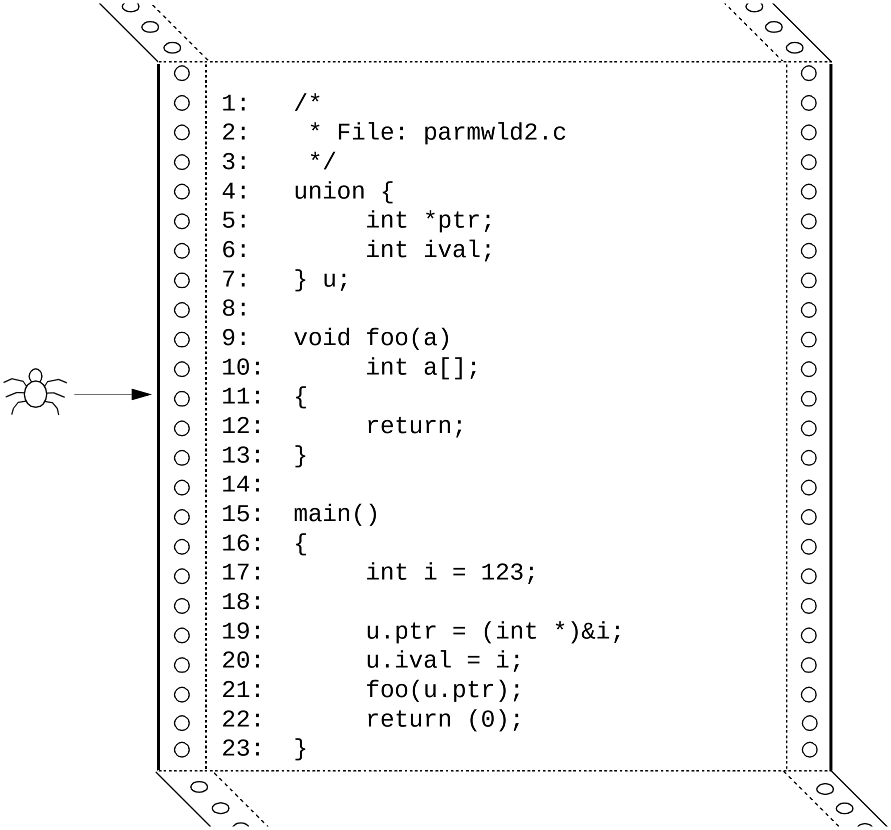
```
[parmwld1.c:6] **PARM_WILD**
>> {

     Array parameter is wild: a

     Pointer : 0x0000007b

          Stack trace where the error occurred:
                     foo() parmwld1.c, 6
                    main() parmwld1.c, 15
```

- Source line at which the problem was detected.
- Description of the problem and the name of the parameter that is in error.
- Value of the bad pointer.
- Stack trace showing the function call sequence leading to the error.

Note that most compilers will generate warning messages for this error since the assignment uses incompatible types.

Problem #2

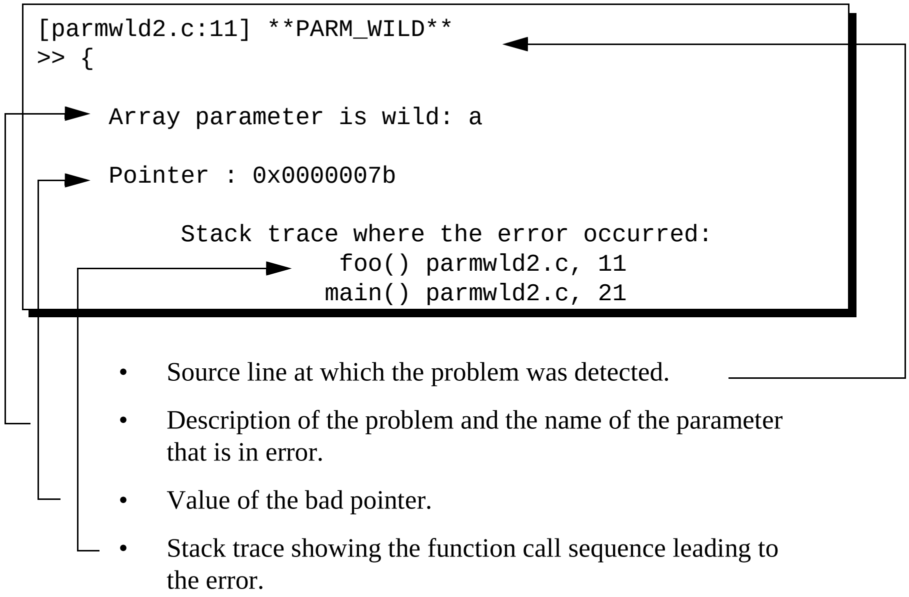
A more insidious version of the same problem can occur when using `union` types. The following code first assigns the pointer element of a union but then overwrites it with another element before finally passing it to a function.



```
/*
 * File: parmwld2.c
 */
union {
     int *ptr;
     int ival;
} u;

void foo(a)
     int a[];
{
     return;
}

main()
{
     int i = 123;

     u.ptr = (int *)&i;
     u.ival = i;
     foo(u.ptr);
     return (0);
}
```

Note that this code will not generate compile time errors.

Diagnosis (at runtime)



```
[parmwld2.c:11] **PARM_WILD**
>> {

        Array parameter is wild: a

        Pointer : 0x0000007b

                Stack trace where the error occurred:
                          foo() parmwld2.c, 11
                         main() parmwld2.c, 21
```

- Source line at which the problem was detected.
- Description of the problem and the name of the parameter that is in error.
- Value of the bad pointer.
- Stack trace showing the function call sequence leading to the error.

Repair

This problem is most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively you can stop the program at some point prior to the error and single-step through the code leading up to the problem

Note that wild pointers can also be generated when Insure++ has only partial information about your program’s structure. This issue is discussed extensively in “Interfaces” on page 103.

READ_BAD_INDEX

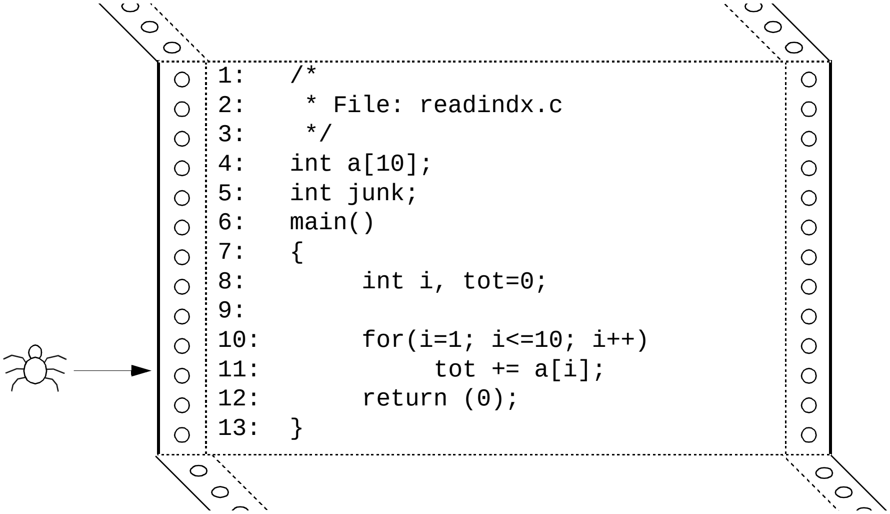
Reading array out of range

This error is generated whenever an illegal value will be used to index an array. It is a particularly common error that can be very difficult to detect, especially if the out-of-range elements happen to have zero values.

If this error can be detected during compilation, an error will be issued instead of the normal runtime error.

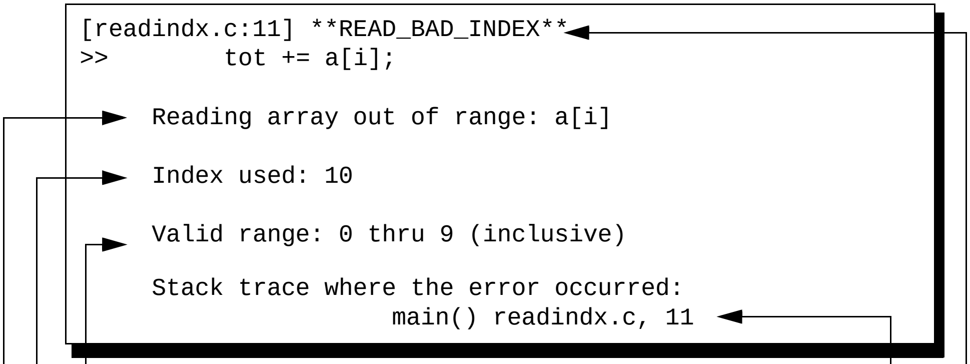
Problem

This code attempts to access an illegal array element due to an incorrect loop range.



```
/*
 * File: readindx.c
 */
int a[10];
int junk;
main()
{
     int i, tot=0;

     for(i=1; i<=10; i++)
          tot += a[i];
     return (0);
}
```

Diagnosis (at runtime)



```
[readindx.c:11] **READ_BAD_INDEX**
>>        tot += a[i];

     Reading array out of range: a[i]

     Index used: 10

     Valid range: 0 thru 9 (inclusive)

     Stack trace where the error occurred:
                      main() readindx.c, 11
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Illegal index value used.
- Valid index range for this array.
- Stack trace showing the function call sequence leading to the error.

Repair

Typical sources of this error include loops with incorrect initial or terminal conditions, as in this example, for which the corrected code is:

```
main()
{
      int i, tot=0, a[10];

      for(i=0; i<sizeof(a)/sizeof(a[0]); i++)
          tot += a[i];
      return (0);
}
```

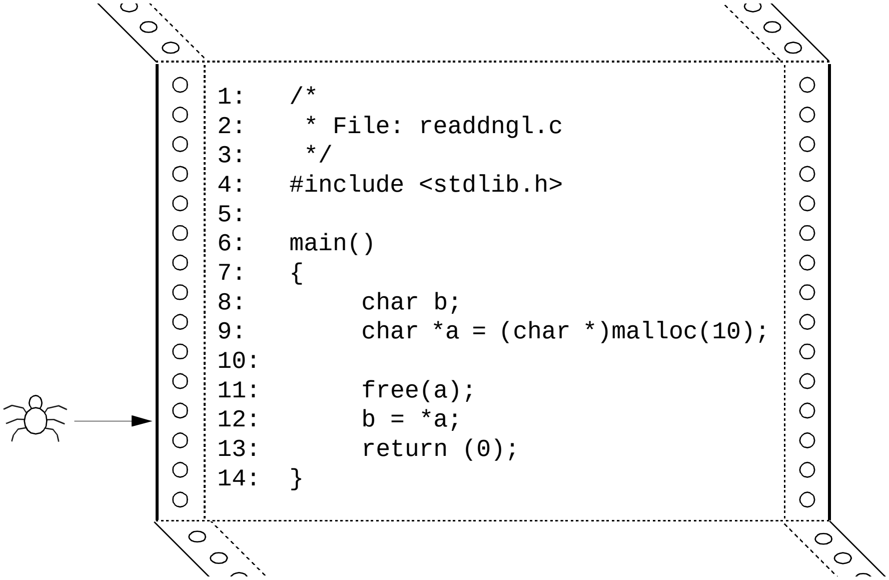
READ_DANGLING

Reading from a dangling pointer

This problem occurs when an attempt is made to dereference a pointer that points to a block of memory that has been freed.

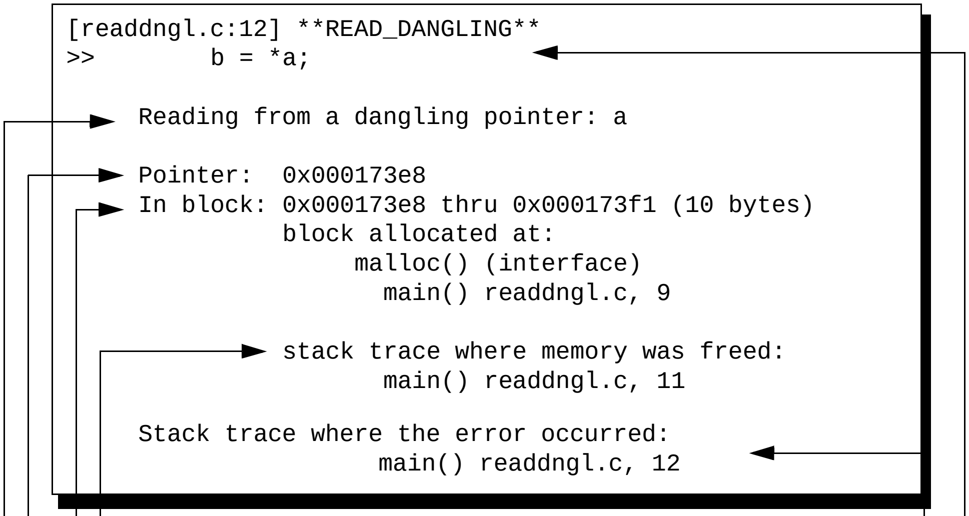
Problem

This code attempts to use a piece of dynamically allocated memory after it has already been freed.



```
/*
 * File: readdngl.c
 */
#include <stdlib.h>

main()
{
     char b;
     char *a = (char *)malloc(10);

     free(a);
     b = *a;
     return (0);
}
```

Diagnosis (at runtime)



```
[readdngl.c:12] **READ_DANGLING**
>>        b = *a;

      Reading from a dangling pointer: a

      Pointer:  0x000173e8
      In block: 0x000173e8 thru 0x000173f1 (10 bytes)
                block allocated at:
                     malloc() (interface)
                       main() readdngl.c, 9

                stack trace where memory was freed:
                       main() readdngl.c, 11

      Stack trace where the error occurred:
                      main() readdngl.c, 12
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Value of the dangling pointer variable.
- Description of the block to which this pointer used to point, including its size, name and the line at which it was allocated.
- Stack trace showing where this block was freed.
- Stack trace showing the function call sequence leading to the error.

Repair

Check that the de-allocation that occurs at the indicated location should, indeed, have taken place. Also check that the pointer you are using should really be pointing to a block allocated at the indicated place.

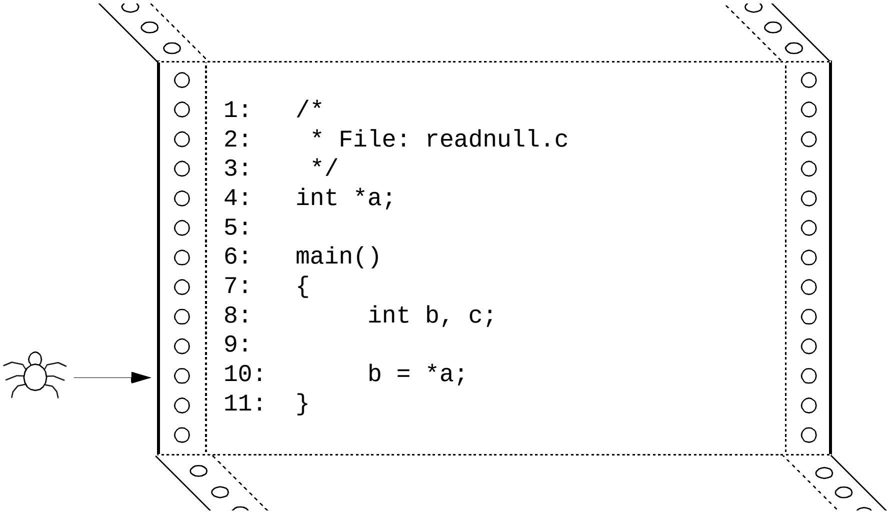
READ_NULL

Reading NULL pointer

This error is generated whenever an attempt is made to dereference a NULL pointer.

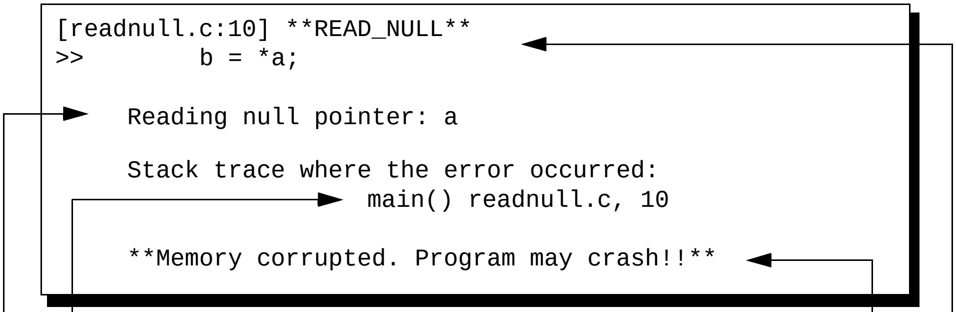
Problem

This code attempts to use a pointer which has not been explicitly initialized. Since the variable a is global, it is initialized to zero by default, which results in dereferencing a NULL pointer in line 10.



```
1:   /*
2:     * File: readnull.c
3:     */
4:   int *a;
5:
6:   main()
7:   {
8:        int b, c;
9:
10:       b = *a;
11:  }
```

Diagnosis (at runtime)



```
[readnull.c:10] **READ_NULL**
>>        b = *a;

     Reading null pointer: a

     Stack trace where the error occurred:
                      main() readnull.c, 10

     **Memory corrupted. Program may crash!!**
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Stack trace showing the function call sequence leading to the error.
- Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

A common cause of this problem is the one shown in the example - use of a pointer that has not been assigned and which is initialized to zero. This is usually due to the omission of an assignment or allocation statement which would give the pointer a reasonable value.

The example code might, for example, be corrected as follows

```
/*
 * File: readnull.c (modified)
 */
int *a;
```

```

main()
{
    int b, c;

    a = &c;
    b = *a;
}
```

A second common source of this error is code which dynamically allocates memory, but then zeroes pointers as blocks are freed. In this case, the error would indicate reuse of a freed block.

A final common problem is caused when one of the dynamic memory allocation routines, `malloc`, `calloc`, or `realloc`, fails and returns a `NULL` pointer. This can happen either because your program passes bad arguments or simply because it asks for too much memory. A simple way of finding this problem with Insure++ is to enable the `RETURN_FAILURE` error code (see page 342) via your `.psrc` file and run the program again. It will then issue diagnostic messages every time a system call fails, including the memory allocation routines.

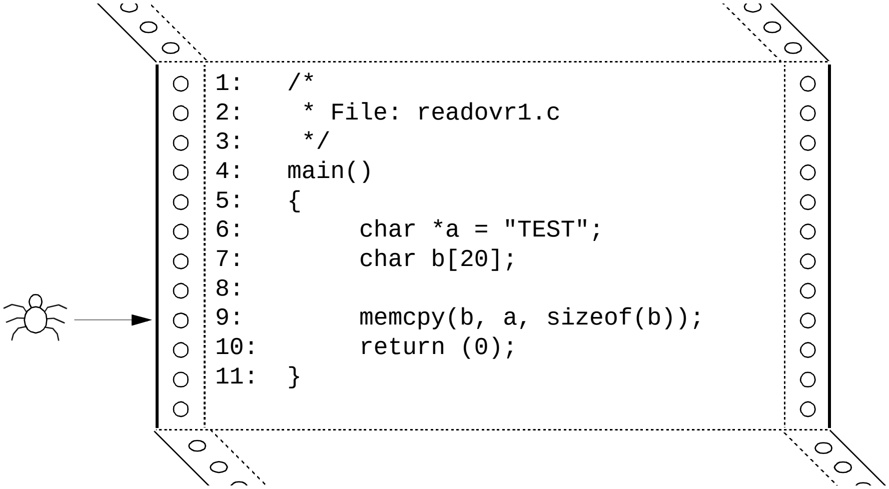
READ_OVERFLOW

Reading overflows memory

This error is generated whenever a read operation would access a piece of memory beyond the valid range for a block.

Problem #1

This code attempts to copy a string into the array `b`. Note that although the array is large enough, the `memcpy` operation will fail, since it attempts to read past the end of the string `a`.



```
/*
 * File: readovr1.c
 */
main()
{
          char *a = "TEST";
          char b[20];

          memcpy(b, a, sizeof(b));
          return (0);
}
```

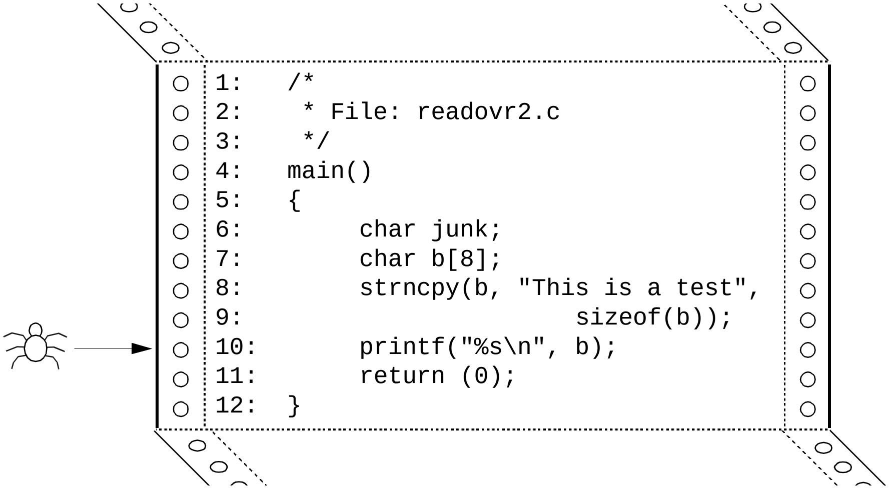
Diagnosis (at runtime)

```
[readovr1.c:9] **READ_OVERFLOW**
>>          memcpy(b, a, sizeof(b));

     Reading overflows memory: <argument 2>

          bbbbb
          | 5 |    15    |
          rrrrrrrrrrrrrrrr

     Reading (r): 0x00012218 thru 0x0001222b (20 bytes)
     From block(b): 0x00012218 thru 0x0001221c (5 bytes)
                    a, declared at readovr1.c, 6

     Stack trace where the error occurred:
                   memcpy() (interface)
                     main() readovr1.c, 9
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Schematic showing the relative layout of the actual memory block (b) and region being read (r). (See “Overflow diagrams” on page 169.)
- Range of memory being read and description of the block from which the read is taking place, including its size and the location of its declaration.
- Stack trace showing the function call sequence leading to the error.

Problem #2

A second fairly common case arises when strings are not terminated properly. The code shown below copies a string using the strncpy routine, which leaves it non-terminated since the buffer is too short. When we attempt to print this message, an error results.



```
/*
 * File: readovr2.c
 */
main()
{
     char junk;
     char b[8];
     strncpy(b, "This is a test",
                    sizeof(b));
     printf("%s\n", b);
     return (0);
}
```

Diagnosis (at runtime)

```
[readovr2.c:10] **READ_OVERFLOW**
>>        printf("%s\n", b);

     String is not null terminated within range: b

     Reading   : 0xf7fffb50
     From block: 0xf7fffb50 thru 0xf7fffb57 (8 bytes)
                 b, declared at readovr2.c, 7

     Stack trace where the error occurred:
                      main() readovr2.c, 10
```

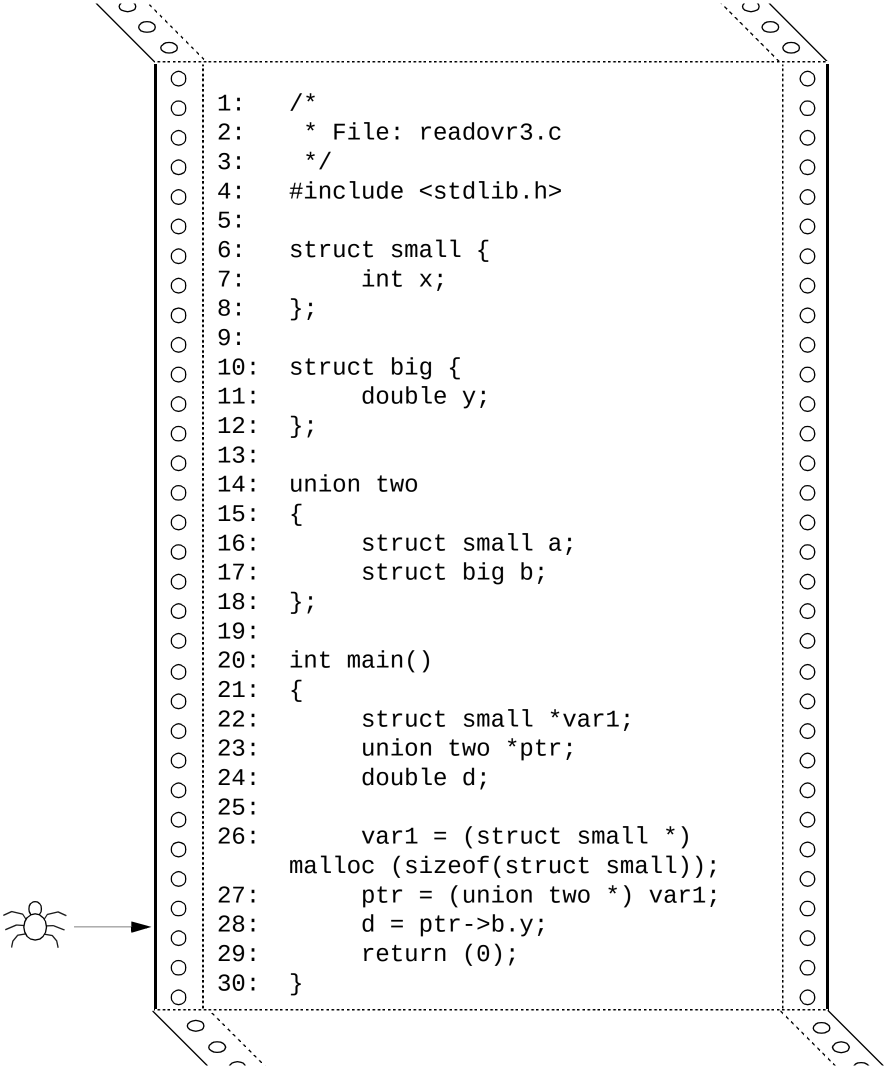
- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Pointer being used as a string.
- Block from which the read is taking place, including its size and the location of its declaration.
- Stack trace showing the function call sequence leading to the error.

A slight variation on this misuse of strings occurs when the pointer, passed as a string, lies completely outside the range of its buffer. In this case, the diagnostics will appear as above except that the description line will contain the message

```
Alleged string does not begin within legal range
```

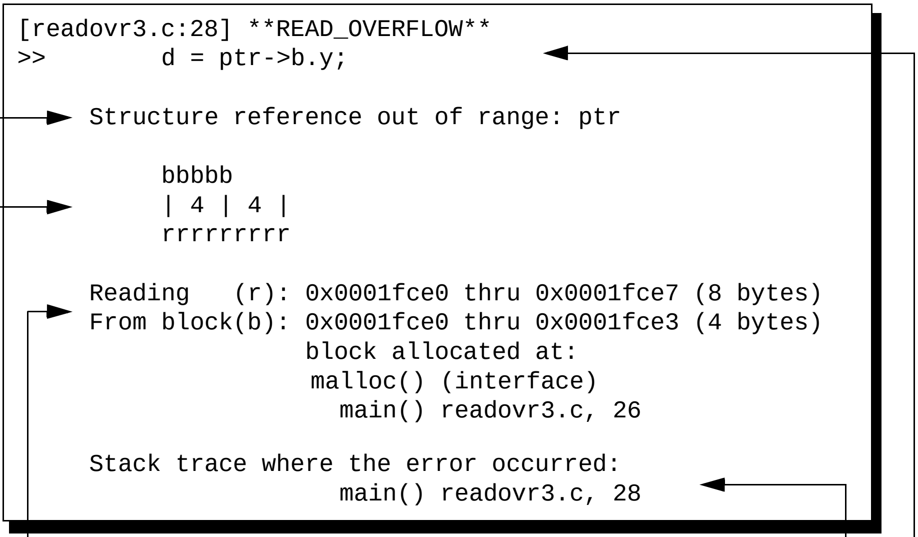
Problem #3

This code attempts to read past the end of the allocated memory block by reading the second element of the union.



```
/*
 * File: readovr3.c
 */
#include <stdlib.h>

struct small {
     int x;
};

struct big {
     double y;
};

union two
{
     struct small a;
     struct big b;
};

int main()
{
     struct small *var1;
     union two *ptr;
     double d;

     var1 = (struct small *)
malloc (sizeof(struct small));
     ptr = (union two *) var1;
     d = ptr->b.y;
     return (0);
}
```

Diagnosis (at runtime)

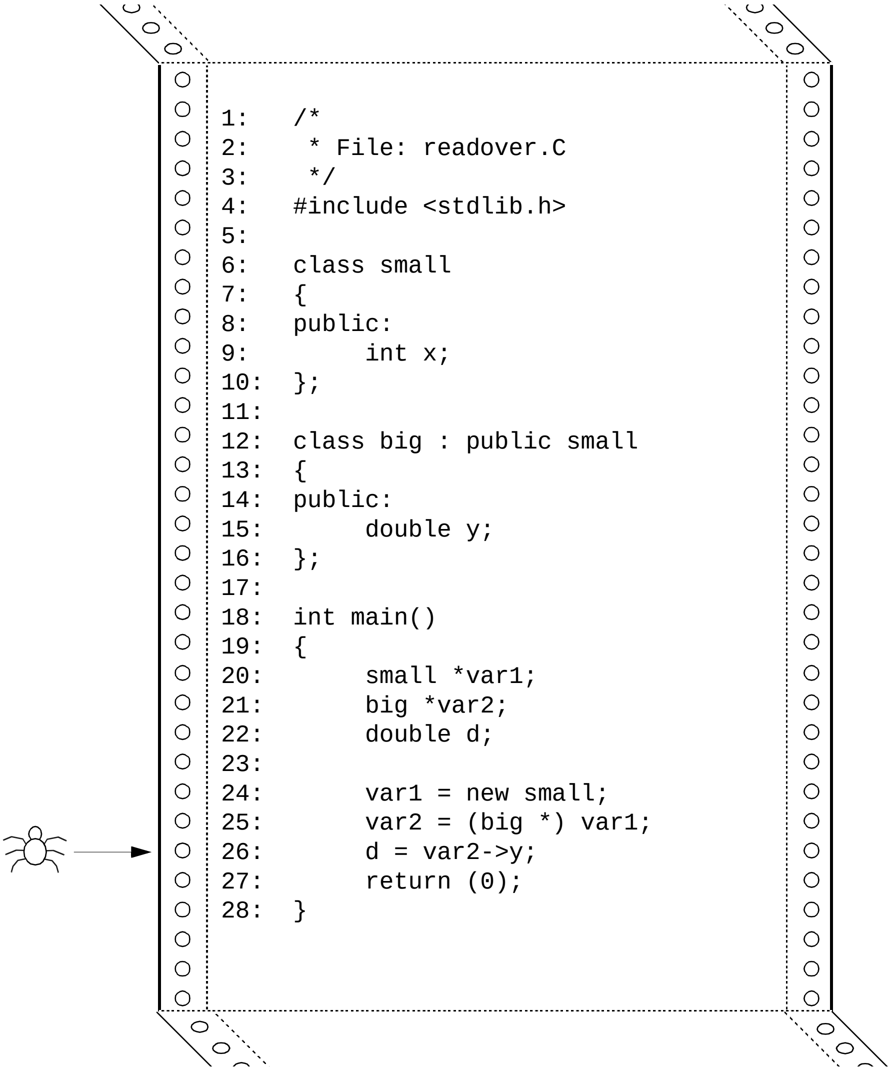


```
[readovr3.c:28] **READ_OVERFLOW**
>>        d = ptr->b.y;

     Structure reference out of range: ptr

          bbbbb
          | 4 | 4 |
          rrrrrrrrr

     Reading   (r): 0x0001fce0 thru 0x0001fce7 (8 bytes)
     From block(b): 0x0001fce0 thru 0x0001fce3 (4 bytes)
                    block allocated at:
                    malloc() (interface)
                      main() readovr3.c, 26

     Stack trace where the error occurred:
                      main() readovr3.c, 28
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Schematic showing the relative layout of the actual memory block (b) and region being read (r). (See “Overflow diagrams” on page 169.)
- Range of memory being read and description of the block from which the read is taking place, including its size and the location of its declaration.
- Stack trace showing the function call sequence leading to the error.

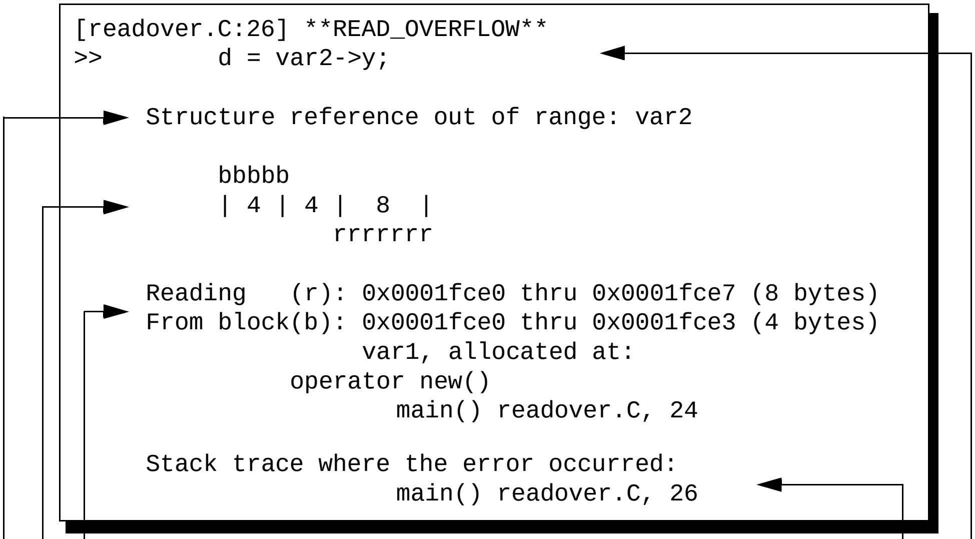
Problem #4

This code shows a C++ example that can occur when using inheritance and casting pointers incorrectly.



```
/*
 * File: readover.C
 */
#include <stdlib.h>

class small
{
public:
     int x;
};

class big : public small
{
public:
     double y;
};

int main()
{
     small *var1;
     big *var2;
     double d;

     var1 = new small;
     var2 = (big *) var1;
     d = var2->y;
     return (0);
}
```

Diagnosis (at runtime)



```
[readover.C:26] **READ_OVERFLOW**
>>         d = var2->y;

     Structure reference out of range: var2

          bbbbb
          | 4 | 4 |  8  |
                  rrrrrrr

     Reading   (r): 0x0001fce0 thru 0x0001fce7 (8 bytes)
     From block(b): 0x0001fce0 thru 0x0001fce3 (4 bytes)
                    var1, allocated at:
               operator new()
                      main() readover.C, 24

     Stack trace where the error occurred:
                      main() readover.C, 26
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Schematic showing the relative layout of the actual memory block (b) and region being read (r). (See “Overflow diagrams” on page 169.)
- Range of memory being read and description of the block from which the read is taking place, including its size and the location of its declaration.
- Stack trace showing the function call sequence leading to the error.

Repair

These errors often occur when reading past the end of a string or using the sizeof operator incorrectly. In most cases, the indicated source line contains a simple error.

The code for problem #1 could, for example, be corrected by changing line 9 to

```
memcpy(b, a, strlen(a)+1);
```

READ_UNINIT_MEM

Reading uninitialized memory

The use of uninitialized memory is a difficult problem to isolate, since the effects of the problem may not show up till much later. This problem is complicated by the fact that quite a lot of references to uninitialized memory are harmless.

To deal with these issues, Insure++ distinguishes two sub-categories of the READ_UNINIT_MEM error class

copy This error code is generated whenever an application assigns a variable using an uninitialized value. In itself, this may not be a problem, since the value may be reassigned to a valid value before use or may never be used. This error category is suppressed by default.

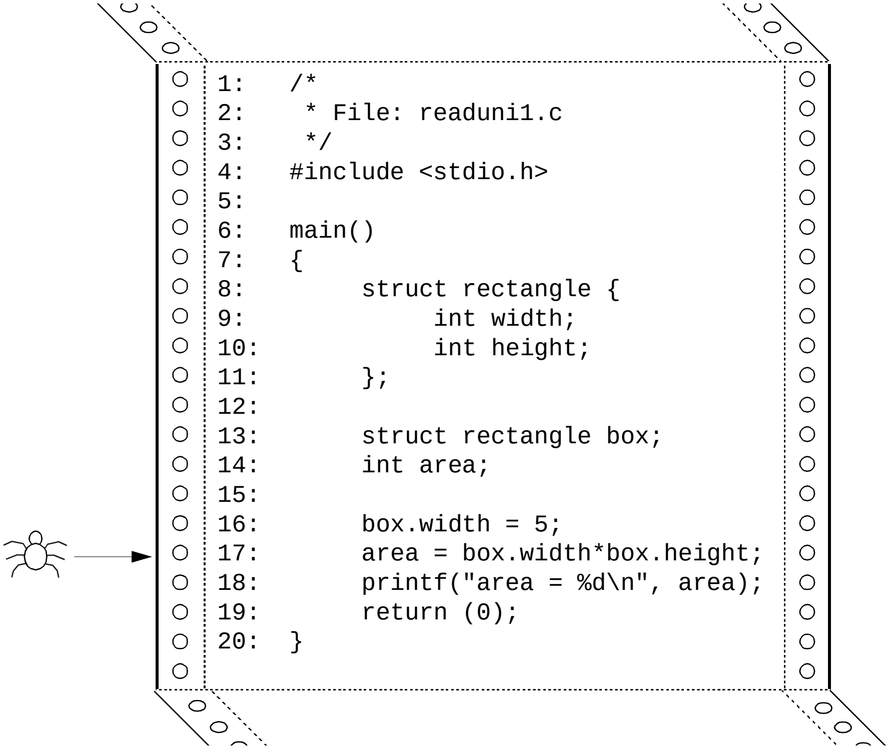
read This code is generated whenever an uninitialized value is used in an expression or some other context where it must be incorrect. This error category is enabled by default, but is detected only if the checking_uninit option is on. (see page 139)

The difference between these two categories is illustrated in the following examples.

Full checking may be disabled by setting the .psrc option "checking_uninit off" (see page 139). If full uninitialized memory checking is disabled, uninitialized pointers will still be detected, but will be reported in the READ_UNINIT_PTR category. (see page 332)

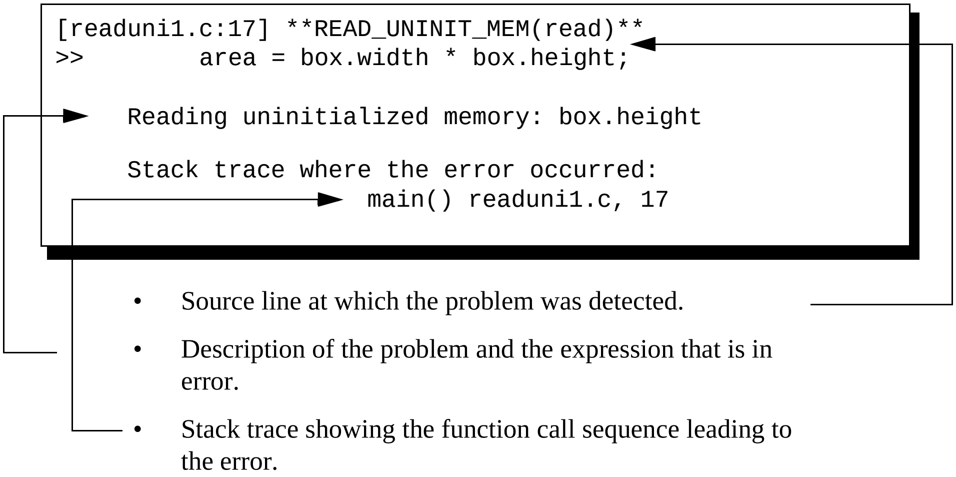
Problem #1

This code attempts to use a structure element which has never been initialized.



```
/*
 * File: readuni1.c
 */
#include <stdio.h>

main()
{
     struct rectangle {
          int width;
          int height;
     };

     struct rectangle box;
     int area;

     box.width = 5;
     area = box.width*box.height;
     printf("area = %d\n", area);
     return (0);
}
```

Diagnosis (at runtime)

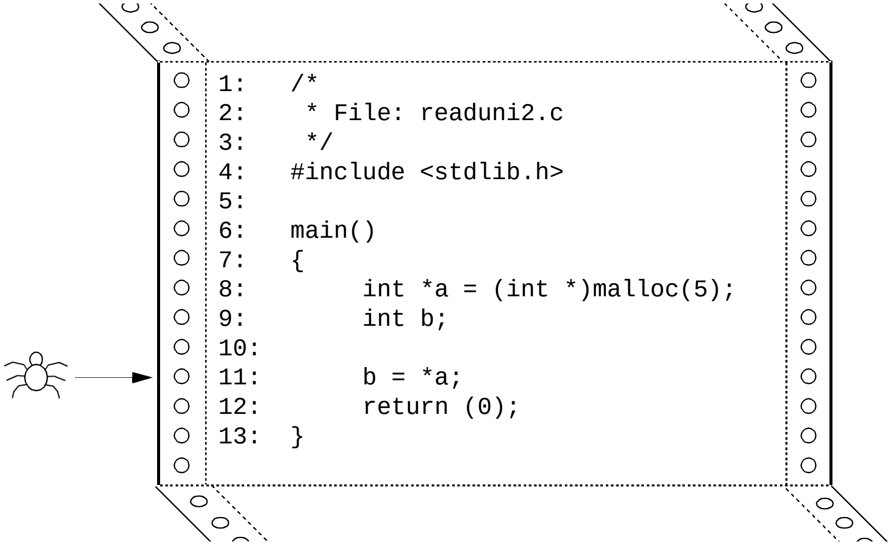


```
[readuni1.c:17] **READ_UNINIT_MEM(read)**
>>        area = box.width * box.height;

     Reading uninitialized memory: box.height

     Stack trace where the error occurred:
                     main() readuni1.c, 17
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Stack trace showing the function call sequence leading to the error.

Problem #2

This code assigns the value `b` using memory returned by the `malloc` system call, which is uninitialized.

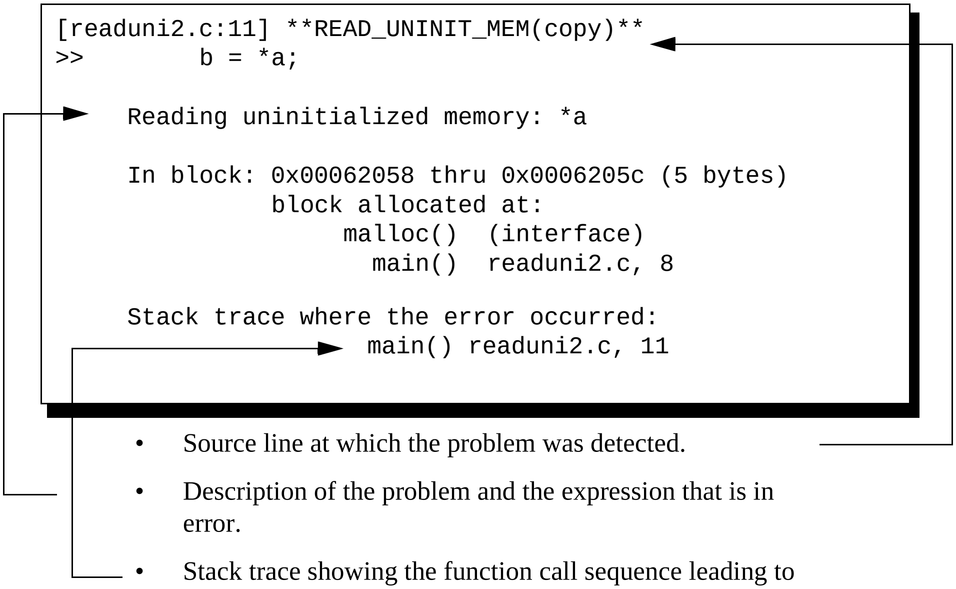


```
1:   /*
2:    * File: readuni2.c
3:    */
4:   #include <stdlib.h>
5:
6:   main()
7:   {
8:        int *a = (int *)malloc(5);
9:        int b;
10:
11:       b = *a;
12:       return (0);
13:  }
```

The code in line 11 of this example falls into the `copy` error sub-category, since the uninitialized value is merely used to assign another variable. If `b` were later used in an expression, it would then generate a `READ_UNINIT_MEM(read)` error.

If the `ints` in lines 8 and 9 of the above example were replaced by `chars`, the error would not be detected by default. To see the error in the new example, you would need to set the `.psrc` option “`checking_uninit_min_size 1`”. For more information about this option, see page 154.

Diagnosis (at runtime)



```
[readuni2.c:11] **READ_UNINIT_MEM(copy)**
>>        b = *a;

     Reading uninitialized memory: *a

     In block: 0x00062058 thru 0x0006205c (5 bytes)
               block allocated at:
                    malloc()  (interface)
                      main()  readuni2.c, 8

     Stack trace where the error occurred:
                     main() readuni2.c, 11
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Stack trace showing the function call sequence leading to the error.

Repair

As mentioned earlier, the READ_UNINIT_MEM(copy) error category is suppressed by default, so you will normally only see errors in the read category. In many cases, these will be errors that can be simply corrected by initializing the appropriate variables. In other cases, these values will have been assigned from other uninitialized variables, which can be detected by unsuppressing the copy sub-category and running again.

READ_UNINIT_PTR

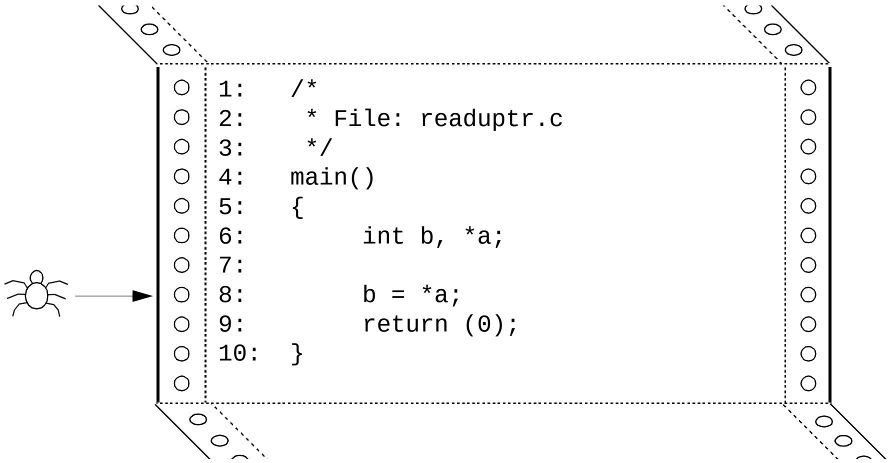
Reading from uninitialized pointer

This error is generated whenever an uninitialized pointer is dereferenced.

This error category will be disabled if full uninitialized memory checking is in effect (the default). In this case, errors are detected in the READ_UNINIT_MEM category instead. (see page 327)

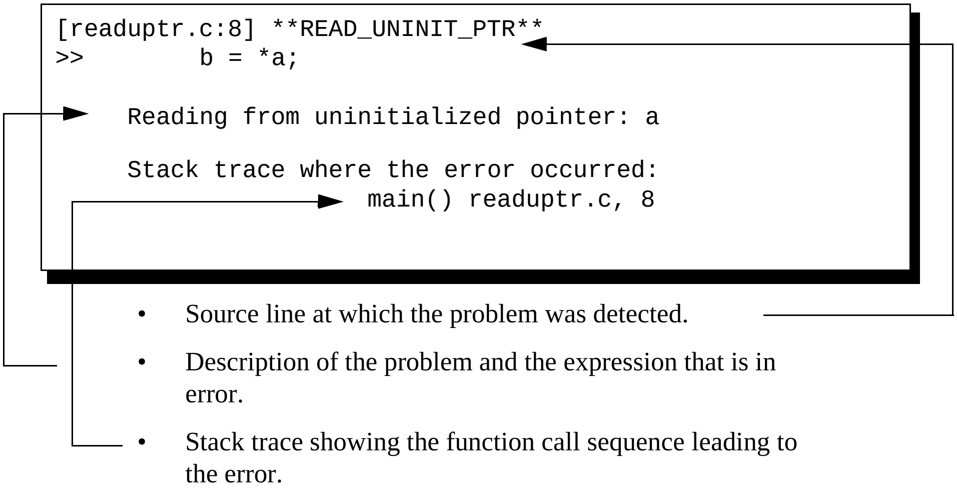
Problem

This code attempts to use the value of the pointer a, even though it has never been initialized.



```
/*
 * File: readuptr.c
 */
main()
{
     int b, *a;

     b = *a;
     return (0);
}
```

Diagnosis (at runtime)



```
[readuptr.c:8] **READ_UNINIT_PTR**
>>        b = *a;

     Reading from uninitialized pointer: a

     Stack trace where the error occurred:
                      main() readuptr.c, 8
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Stack trace showing the function call sequence leading to the error.

Repair

This problem is usually caused by omitting an assignment or allocation statement that would initialize a pointer. The code given, for example, could be corrected by including an assignment as shown below.

```
/*
 * File: readuptr.c (Modified)
 */
main()
{
      int b, *a, c;

      a = &c;
      b = *a;
      return (0);
}
```

READ_WILD

Reading wild pointer

This problem occurs when an attempt is made to dereference a pointer whose value is invalid or which Insure++ did not see allocated.

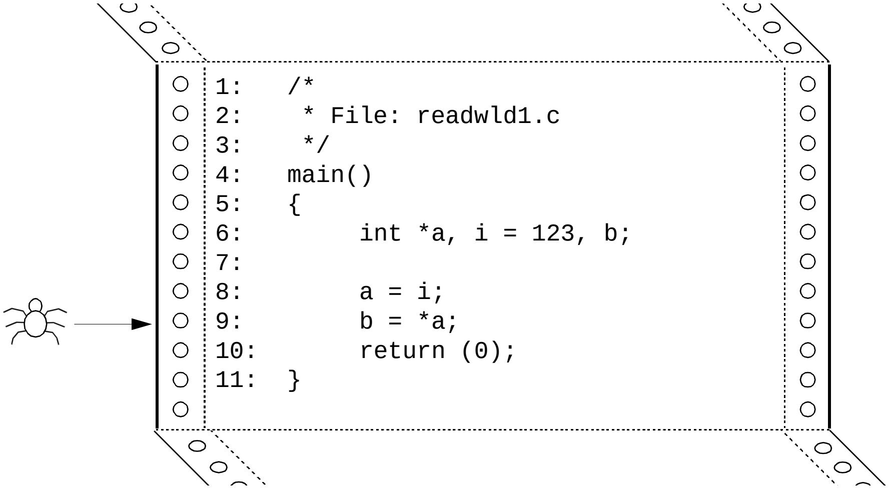
This can come about in several ways:

- Errors in user code that result in pointers that don’t point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

This discussion centers on the first type of problem described here. A detailed discussion of the second topic, including samples of its generation and repair can be found in “Interfaces” on page 103.

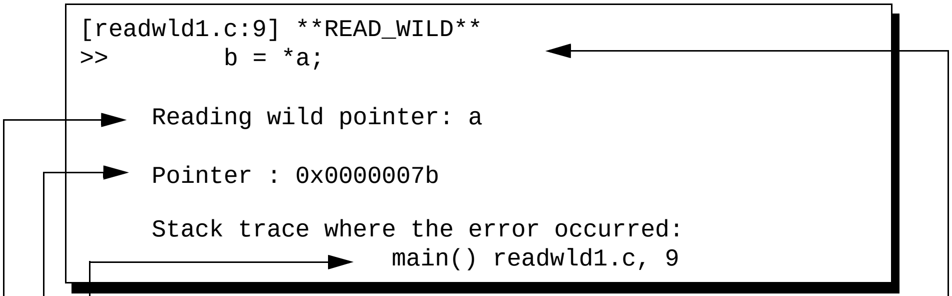
Problem #1

The following code attempts to use the address of a variable but contains an error at line 8 - the address operator (&) has been omitted.



```
1:   /*
2:    * File: readwld1.c
3:    */
4:   main()
5:   {
6:        int *a, i = 123, b;
7:
8:        a = i;
9:        b = *a;
10:       return (0);
11:  }
```

Diagnosis (at runtime)



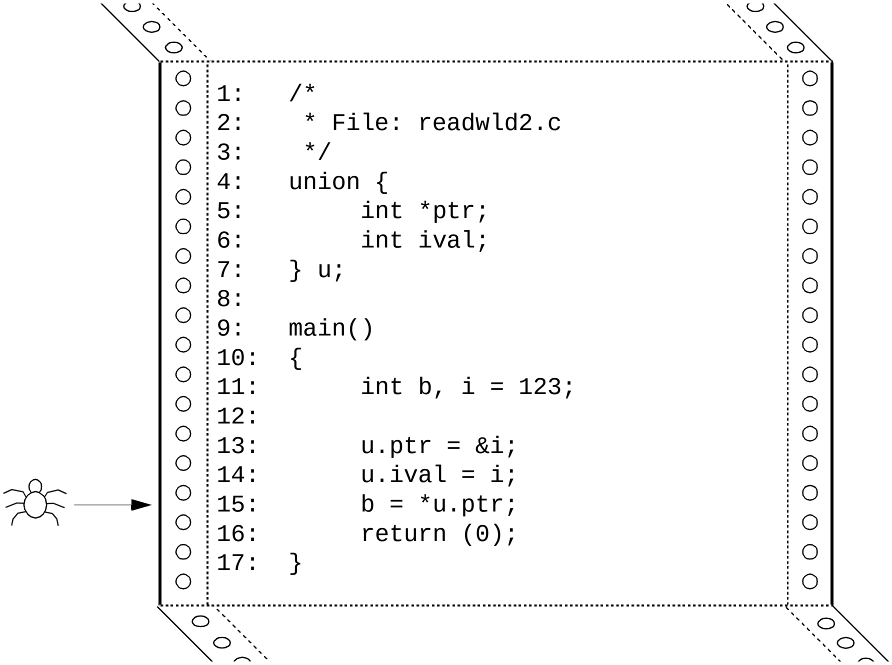
```
[readwld1.c:9] **READ_WILD**
>>        b = *a;

     Reading wild pointer: a

     Pointer : 0x0000007b

     Stack trace where the error occurred:
                      main() readwld1.c, 9
```

- Source line at which the problem was detected.
- Description of the problem and the name of the parameter that is in error.
- Value of the bad pointer.
- Stack trace showing the function call sequence leading to the error.

Note that most compilers will generate warning messages for this error since the assignment uses incompatible types.

Problem #2

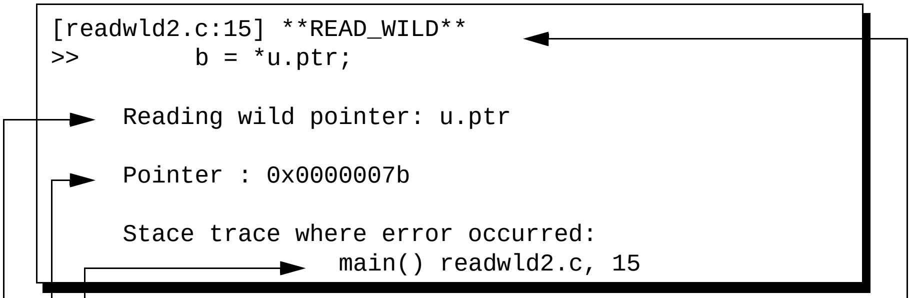
A more insidious version of the same problem can occur when using `union` types. The following code first assigns the pointer element of a union but then overwrites it with another element before using it.



```
/*
 * File: readwld2.c
 */
union {
     int *ptr;
     int ival;
} u;

main()
{
     int b, i = 123;

     u.ptr = &i;
     u.ival = i;
     b = *u.ptr;
     return (0);
}
```

Note that this code will not generate compile time errors.

Diagnosis (at runtime)



```
[readwld2.c:15] **READ_WILD**
>>        b = *u.ptr;

     Reading wild pointer: u.ptr

     Pointer : 0x0000007b

     Stace trace where error occurred:
                    main() readwld2.c, 15
```

- Source line at which the problem was detected.
- Description of the problem and the name of the parameter that is in error.
- Value of the bad pointer.
- Stack trace showing the function call sequence leading to the error.

Repair

The simpler types of problem are most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively you can stop the program at some point shortly before the error and single-step through the code leading up to the problem.

Note that wild pointers can also be generated when Insure++ has only partial information about your program’s structure. This issue is discussed extensively in “Interfaces” on page 103.

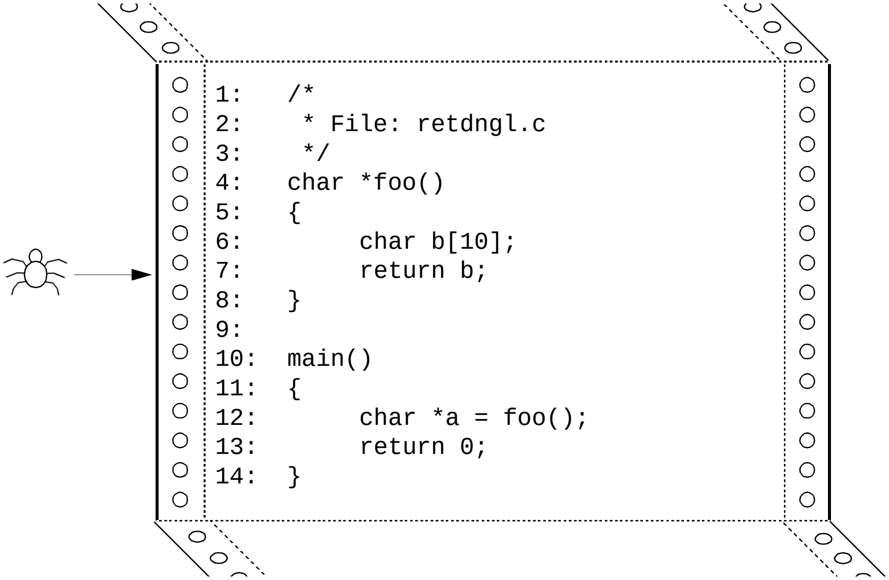
RETURN_DANGLING

Returning pointer to local variable

This error is generated whenever a function returns a pointer to a (non-static) local variable. Since the stack frame of this routine will disappear when the function returns, this pointer is never valid.

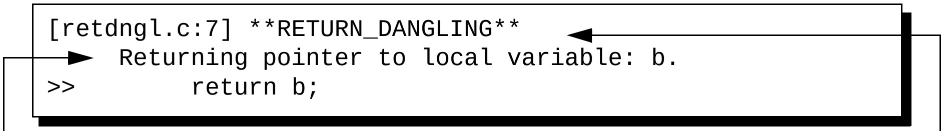
Problem

The following code shows the routine `foo` returning a pointer to a local variable.



```
/*
 * File: retdngl.c
 */
char *foo()
{
     char b[10];
     return b;
}

main()
{
     char *a = foo();
     return 0;
}
```

Diagnosis (during compilation)



```
[retdngl.c:7] **RETURN_DANGLING**
     Returning pointer to local variable: b.
>>          return b;
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.

Repair

The pointer returned in this manner can be made legal in one of several ways.

- Declaring the memory block `static` in the called routine, i.e., line 6 would become

```
static char b[10];
```

- Allocating a block dynamically instead of on the stack and returning a pointer to it, e.g.

```
char *foo()
{
    return malloc(10);
}
```

- Making the memory block into a global variable rather than a local one.

Occasionally, the value returned from the function is never used in which case it is safest to change the declaration of the routine to indicate that no value is returned.

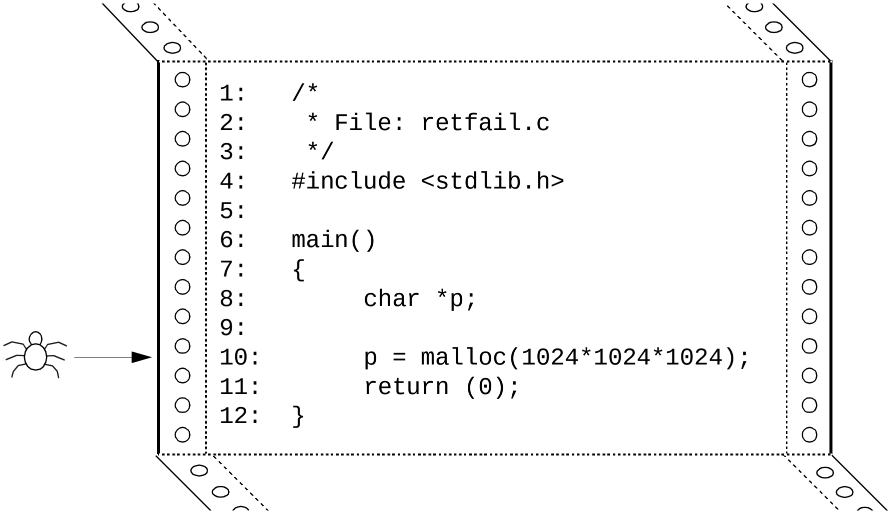
RETURN_FAILURE

Function call returned an error

A particularly difficult problem to track with conventional methods is that of incorrect return code from system functions. Insure++ is equipped with interface definitions for system libraries that enable it to check for errors when functions are called. Normally, these messages are suppressed, since applications often include their own handling for system calls that return errors. In some cases, however, it may be useful to enable these messages to track down totally unexpected behavior.

Problem

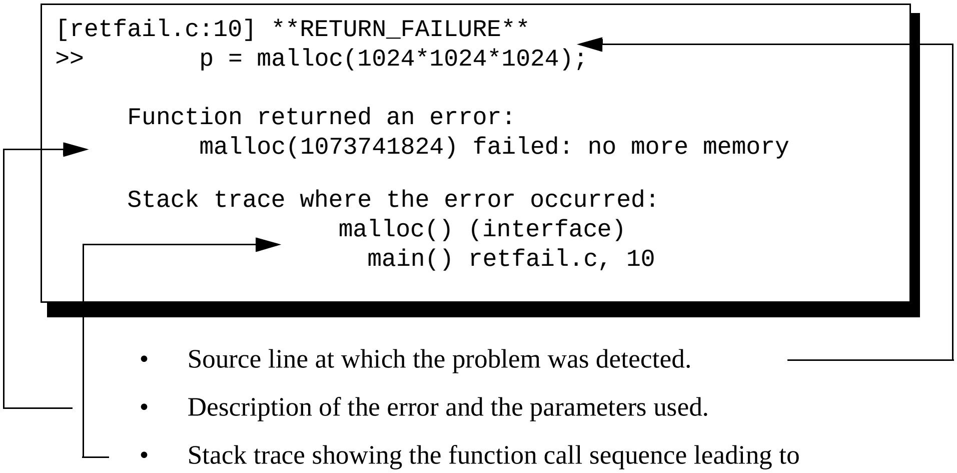
A particularly common problem occurs when applications run out of memory as in the following code.



```
/*
 * File: retfail.c
 */
#include <stdlib.h>

main()
{
     char *p;

     p = malloc(1024*1024*1024);
     return (0);
}
```

Diagnosis

Normally, this code will run without displaying any messages. If RETURN_FAILURE messages are enabled, however, the following display will result.



```
[retfail.c:10] **RETURN_FAILURE**
>>         p = malloc(1024*1024*1024);

     Function returned an error:
          malloc(1073741824) failed: no more memory

     Stack trace where the error occurred:
                    malloc() (interface)
                      main() retfail.c, 10
```

- Source line at which the problem was detected.
- Description of the error and the parameters used.
- Stack trace showing the function call sequence leading to the error.

Repair

These messages are normally suppressed, but can be enabled by adding the option

```
insure++.unsuppress RETURN_FAILURE
```

to your .psrc file.

RETURN_INCONSISTENT

Function has inconsistent return type

Insure++ checks that each function returns a result consistent with its declared data type, and that a function with a declared return type actually returns an appropriate value.

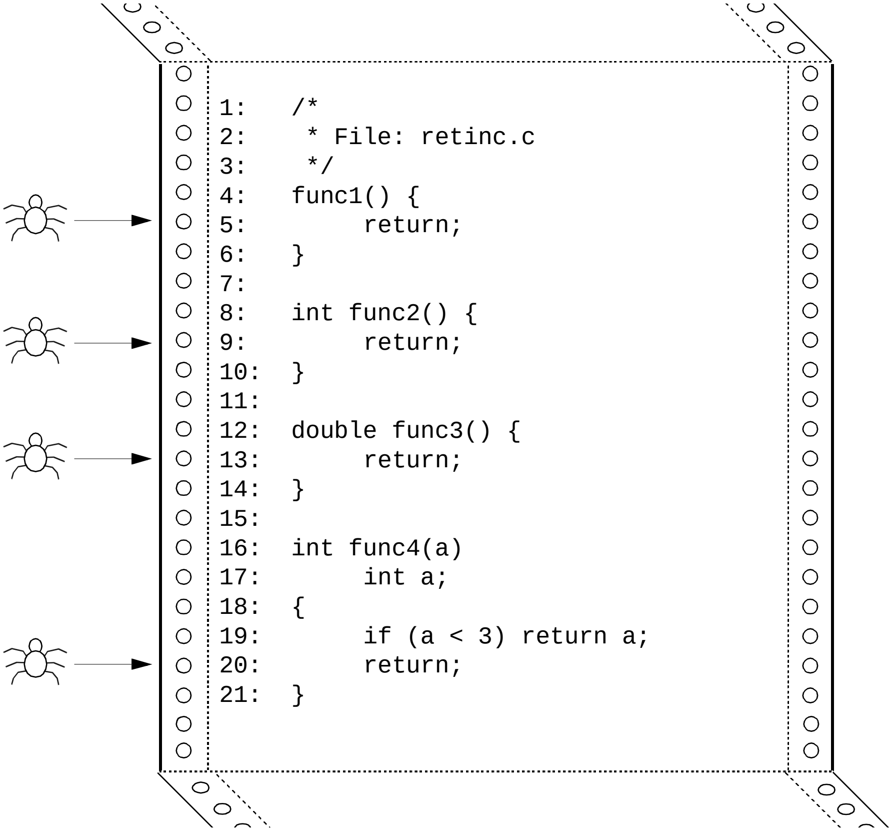
Because there are several different ways in which functions and return values can be declared, Insure++ divides up this error category into four levels or subcategories as follows:

Level 1	Function has no explicitly declared return type (and so defaults to `int`) and returns no value. (This error level is normally suppressed.)
Level 2	Function is explicitly declared to return type `int` but returns nothing.
Level 3	Function explicitly declared to return a data type other than `int` but returns no value.
Level 4	The function returns a value of one type at one statement and another data type at another statement.

In many applications, errors at levels 1 and 2 need to be suppressed, since older codes often include these constructs.

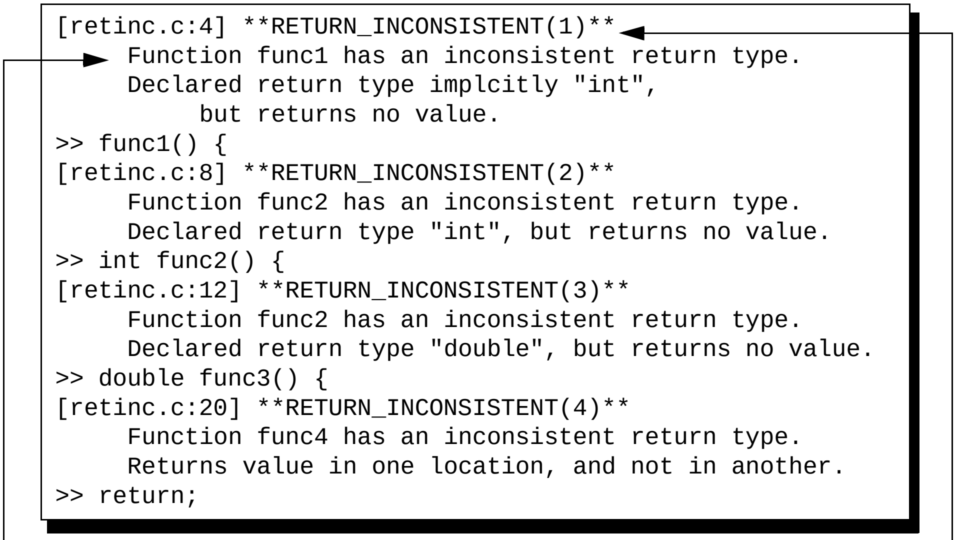
Problem

The following code demonstrates the four different error levels.



```
/*
 * File: retinc.c
 */
func1() {
     return;
}

int func2() {
     return;
}

double func3() {
     return;
}

int func4(a)
     int a;
{
     if (a < 3) return a;
     return;
}
```

Diagnosis (During compilation).



```
[retinc.c:4] **RETURN_INCONSISTENT(1)**
     Function func1 has an inconsistent return type.
     Declared return type implcitly "int",
          but returns no value.
>> func1() {
[retinc.c:8] **RETURN_INCONSISTENT(2)**
     Function func2 has an inconsistent return type.
     Declared return type "int", but returns no value.
>> int func2() {
[retinc.c:12] **RETURN_INCONSISTENT(3)**
     Function func2 has an inconsistent return type.
     Declared return type "double", but returns no value.
>> double func3() {
[retinc.c:20] **RETURN_INCONSISTENT(4)**
     Function func4 has an inconsistent return type.
     Returns value in one location, and not in another.
>> return;
```

- Source line at which the problem was detected.
- Description of the error and the parameters used.

Repair

As already suggested, older codes often generate errors at levels 1 and 2 which are not particularly serious. You can either correct these problems by adding suitable declarations or suppress them by adding the option

```
insure++.suppress RETURN_INCONSISTENT(1, 2)
```

to your `.psrc` file.

Errors at levels 3 and 4 should probably be investigated and corrected.

UNUSED_VAR

Unused variables

Insure++ has the ability to detect unused variables in your code. Since these are not normally errors, but informative messages, this category is disabled by default.

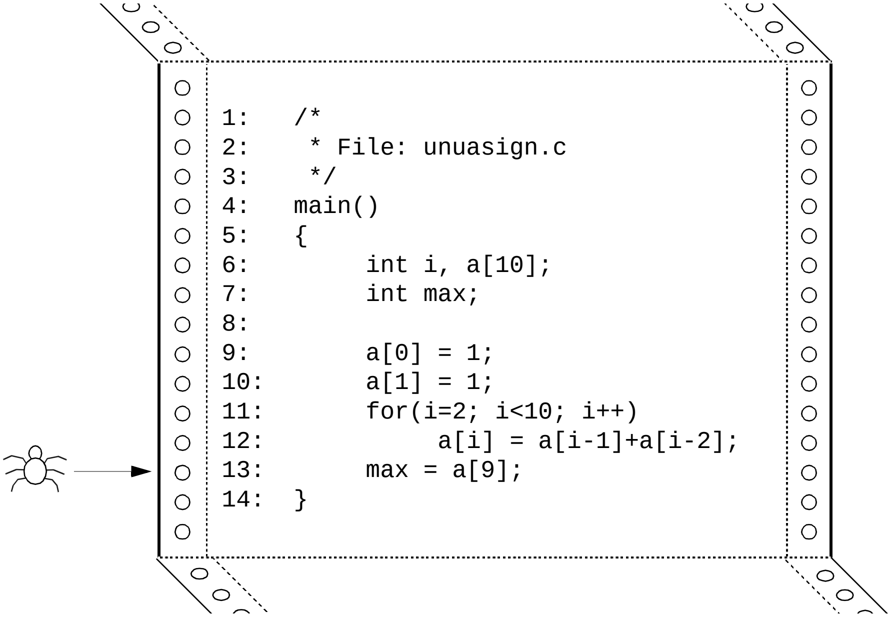
Two different sub-categories are distinguished

`assigned` The variable is assigned a value but never used.

`unused` The variable is never used.

Problem #1

The following code assigns a value to the variable `max` but never uses it.



```
/*
 * File: unuasign.c
 */
main()
{
     int i, a[10];
     int max;

     a[0] = 1;
     a[1] = 1;
     for(i=2; i<10; i++)
          a[i] = a[i-1]+a[i-2];
     max = a[9];
}
```

Diagnosis (during compilation)

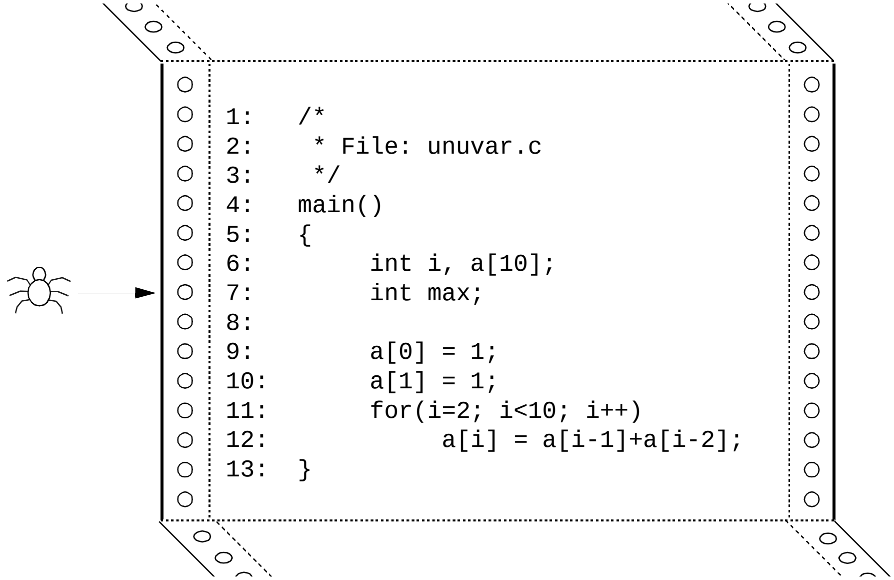
Normally this code will run without displaying any messages. If `UNUSED_VAR` messages are enabled, however, the following display will result.

```
[unuasign.c:7] **UNUSED_VAR(assigned)**
     Variable assigned but never used: max
>>        int max;
```

- Source line at which the problem was detected.
- Description of the error and the parameters used.

Problem #2

The following code never uses the variable `max`.



```
1:   /*
2:    * File: unuvar.c
3:    */
4:   main()
5:   {
6:        int i, a[10];
7:        int max;
8:
9:        a[0] = 1;
10:       a[1] = 1;
11:       for(i=2; i<10; i++)
12:            a[i] = a[i-1]+a[i-2];
13:  }
```

Diagnosis (during compilation)

If `UNUSED_VAR` messages are enabled, however, the following display will result.

```
[unuvar.c:7] **UNUSED_VAR(unused)**
     Variable declared but never used: max
>>        int max;
```

- Source line at which the problem was detected.
- Description of the error and the parameters used.

Repair

These messages are normally suppressed but can be enabled by adding the option

```
insure++.unsuppress UNUSED_VAR
```

to your `.psrc` file.

You can also enable each sub-category independently with an option such as

```
insure++.unsuppress UNUSED_VAR(assigned)
```

In most cases, the corrective action to be taken is to remove the offending statement, since it is not affecting the behavior of the application. In certain circumstances, these errors may denote logical program errors in which a variable should have been used but wasn't.

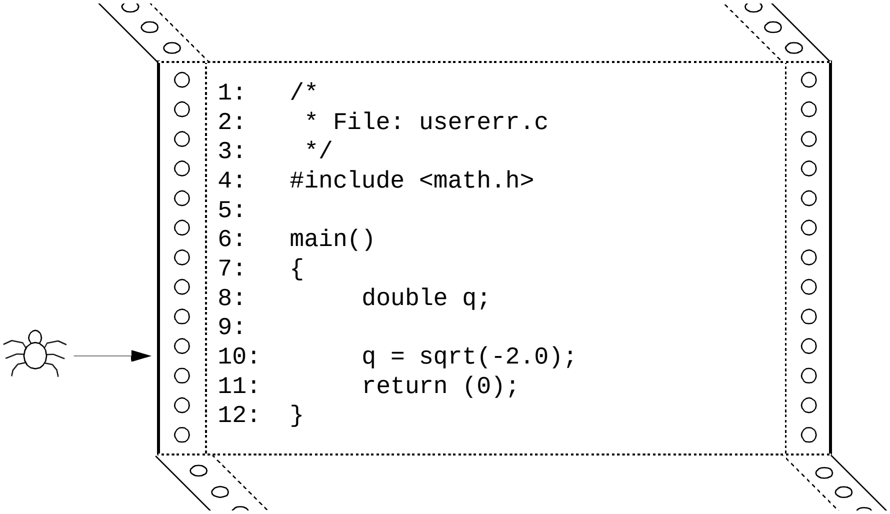
USER_ERROR

User generated error message

This error is generated when a program violates a rule specified in an interface module. These normally check that parameters passed to system level or user functions fall within legal ranges or are otherwise valid. This behavior is different from the RETURN_FAILURE error code, which normally indicates that the call to the function was made with valid data, but that it still returned an error for some, possibly anticipated, reason.

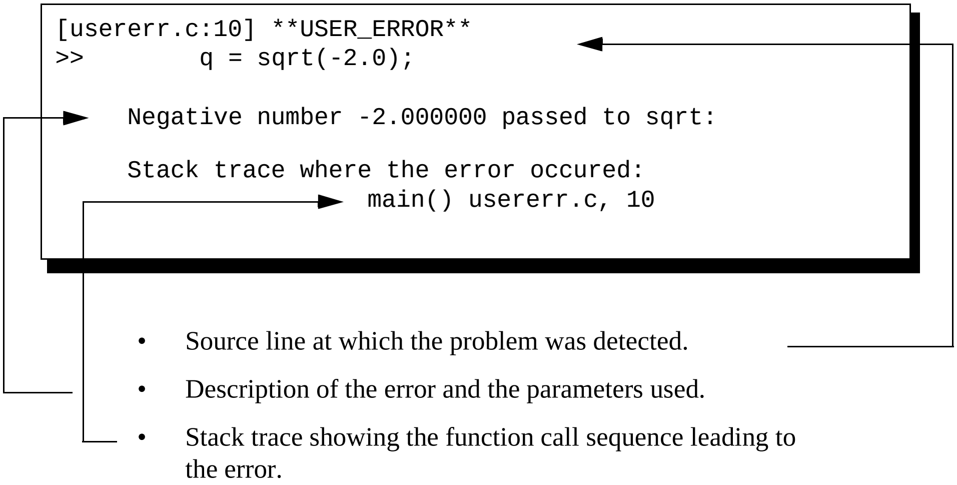
Problem

These problems fall into many different categories. A particularly simple example is shown in the following code, which calls the sqrt function and passes it a negative argument.



```
/*
 * File: usererr.c
 */
#include <math.h>

main()
{
     double q;

     q = sqrt(-2.0);
     return (0);
}
```

Diagnosis (at runtime)



```
[usererr.c:10] **USER_ERROR**
>>        q = sqrt(-2.0);

     Negative number -2.000000 passed to sqrt:

     Stack trace where the error occured:
                      main() usererr.c, 10
```

- Source line at which the problem was detected.
- Description of the error and the parameters used.
- Stack trace showing the function call sequence leading to the error.

Repair

Each message in this category is caused by a different problem, which should be evident from the printed diagnostic. Usually, these checks revolve around the legality of various arguments to functions.

These messages can be suppressed by adding the option

```
insure++.suppress USER_ERROR
```

to your `.psrc` file.

VIRTUAL_BAD

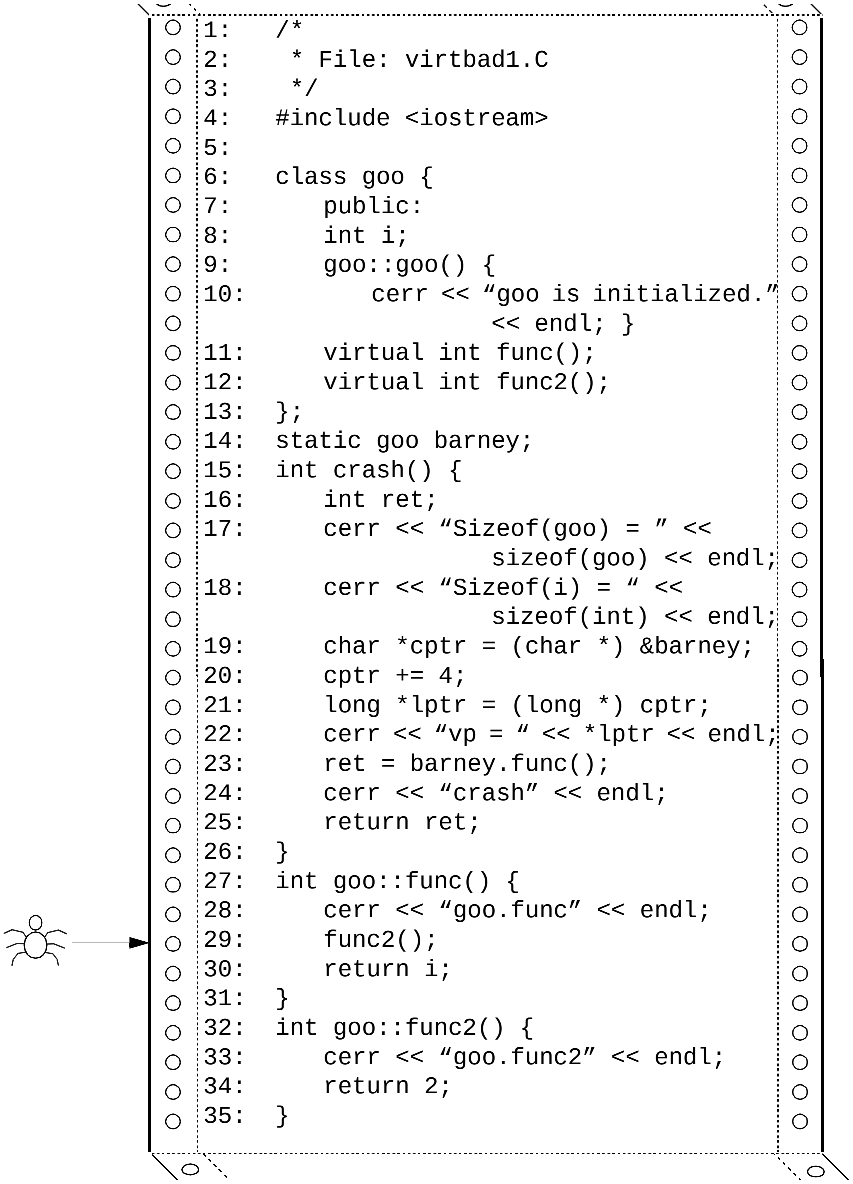
Error in runtime initialization of virtual functions

This error is caused when a virtual function has not been initialized prior to being used by another function.

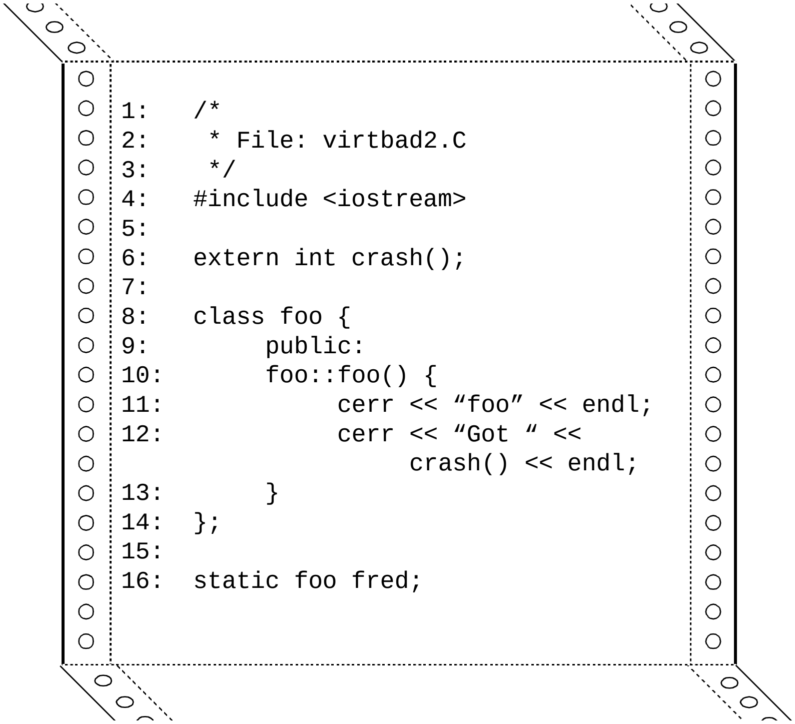
Problem

The following pieces of code illustrate this error. The virtual function `func` is declared in virtbad1.C in the `goo` class. A static variable of this class, `barney`, is also declared in that file. The function crash calls `func` through `barney` in line 23. In file virtbad2.C, a static variable of class `foo`, `fred`, is declared. Class `foo` calls `crash`, which then in turn ends up calling the virtual function `func`. A virtual function's address is not established until the program is initialized at runtime, and static functions are also initialized at runtime. This means that depending on the order of initialization, `fred` could be trying to find `func`, which does not yet have an address. The VIRTUAL_BAD error message is generated when this code is compiled with Insure++.

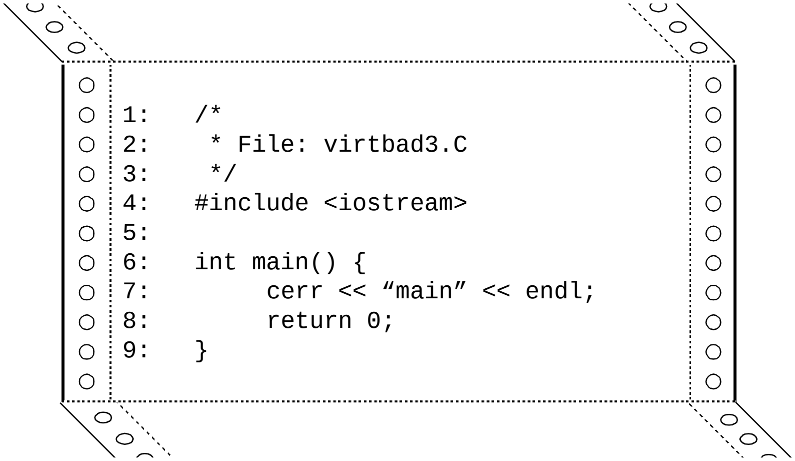
Note: Due to differences in the object layout of different compilers, this error may not be detected with certain compilers.



```
/*
 * File: virtbad1.C
 */
#include <iostream>

class goo {
   public:
   int i;
   goo::goo() {
      cerr << “goo is initialized.”
               << endl; }
   virtual int func();
   virtual int func2();
};
static goo barney;
int crash() {
   int ret;
   cerr << “Sizeof(goo) = ” <<
               sizeof(goo) << endl;
   cerr << “Sizeof(i) = “ <<
               sizeof(int) << endl;
   char *cptr = (char *) &barney;
   cptr += 4;
   long *lptr = (long *) cptr;
   cerr << “vp = “ << *lptr << endl;
   ret = barney.func();
   cerr << “crash” << endl;
   return ret;
}
int goo::func() {
   cerr << “goo.func” << endl;
   func2();
   return i;
}
int goo::func2() {
   cerr << “goo.func2” << endl;
   return 2;
}
```

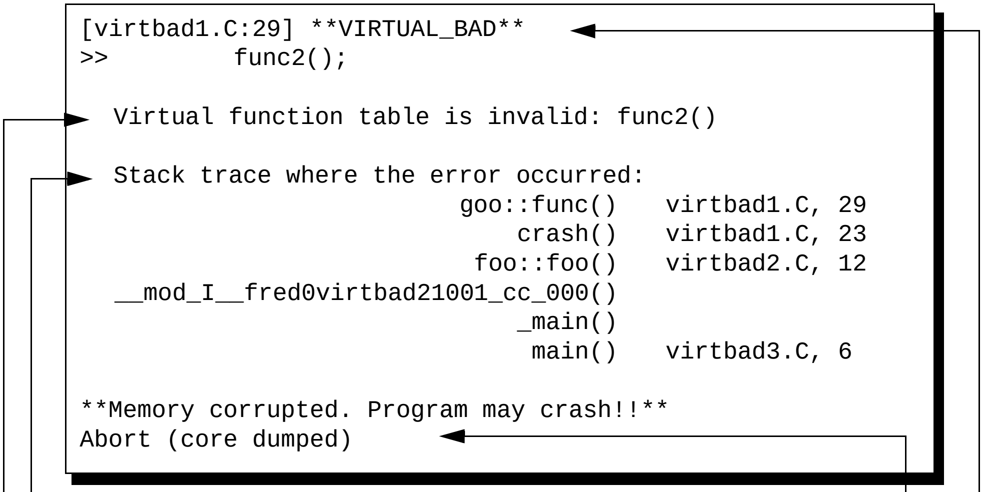


```
/*
 * File: virtbad2.C
 */
#include <iostream>

extern int crash();

class foo {
     public:
     foo::foo() {
          cerr << “foo” << endl;
          cerr << “Got “ <<
               crash() << endl;
     }
};

static foo fred;
```



```
/*
 * File: virtbad3.C
 */
#include <iostream>

int main() {
     cerr << “main” << endl;
     return 0;
}
```

Diagnosis (at runtime)



```
[virtbad1.C:29] **VIRTUAL_BAD**
>>         func2();

  Virtual function table is invalid: func2()

  Stack trace where the error occurred:
                        goo::func()   virtbad1.C, 29
                            crash()   virtbad1.C, 23
                         foo::foo()   virtbad2.C, 12
  __mod_I__fred0virtbad21001_cc_000()
                            _main()
                             main()   virtbad3.C, 6

**Memory corrupted. Program may crash!!**
Abort (core dumped)
```

- Source line at which the problem was detected.
- Description of the problem and which virtual function caused the error.
- Stack trace showing the function call sequence leading to the error.
- Core dumps typically follow these messages, as any usage of the dynamic memory functions will be unable to cope.

Repair

The error in the sample code could be eliminated by not making `fred` static. In that case, the address for `func` would be generated during the initialization before any requests for it existed, and there would be no problems.

WRITE_BAD_INDEX

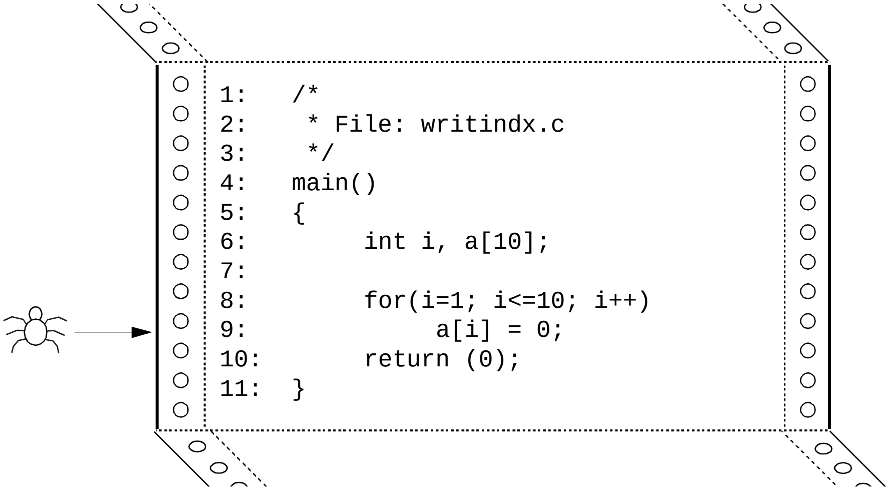
Writing array out of range

This error is generated whenever an illegal value will be used to index an array which is being written.

If this error can be detected during compilation, a compilation error will be issued instead of the normal runtime error.

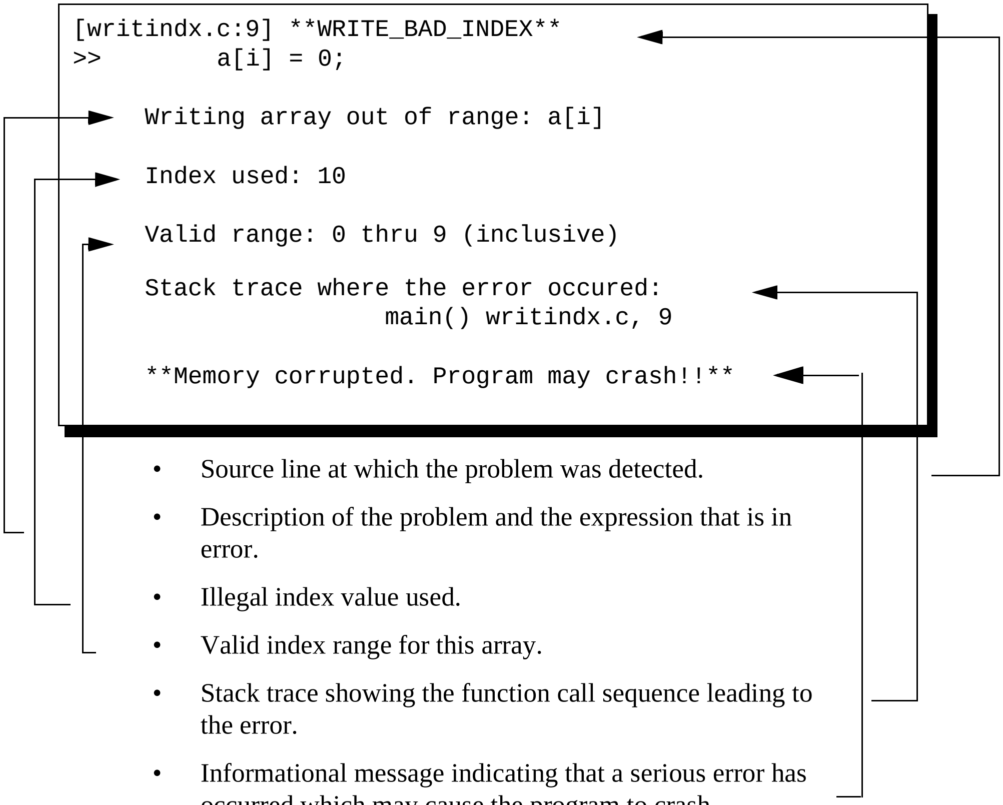
Problem

This code attempts to access an illegal array element due to an incorrect loop range.



```
/*
 * File: writindx.c
 */
main()
{
     int i, a[10];

     for(i=1; i<=10; i++)
          a[i] = 0;
     return (0);
}
```

Diagnosis (at runtime)



```
[writindx.c:9] **WRITE_BAD_INDEX**
>>        a[i] = 0;

      Writing array out of range: a[i]

      Index used: 10

      Valid range: 0 thru 9 (inclusive)

      Stack trace where the error occured:
                       main() writindx.c, 9

      **Memory corrupted. Program may crash!!**
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Illegal index value used.
- Valid index range for this array.
- Stack trace showing the function call sequence leading to the error.
- Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

This is normally a fatal error and is often introduced algorithmically.

One common source of this error is using “stretchy arrays” without telling Insure++ about them. A “stretchy array” is an array whose size is only determined at runtime. For an example as well as an explanation of how to use Insure++ see “Report summaries” on page 72.

Other typical sources include loops with incorrect initial or terminal conditions, as in this example, for which the corrected code is:

```
main()
{
      int i, a[10];

      for(i=; i<sizeof(a)/sizeof(a[0]); i++)
           a[i] = 0;
      return (0);
}
```

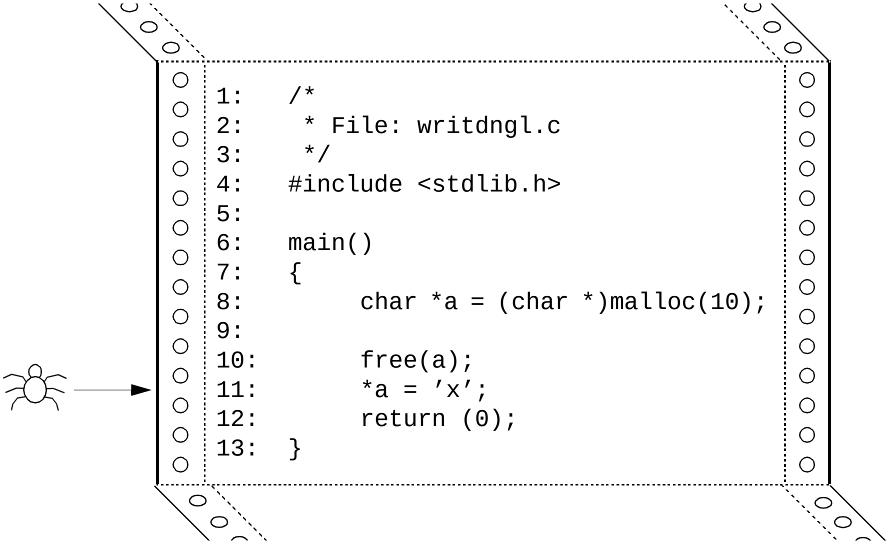
WRITE_DANGLING

Writing to a dangling pointer

This problem occurs when an attempt is made to dereference a pointer that points to a block of memory that has been freed.

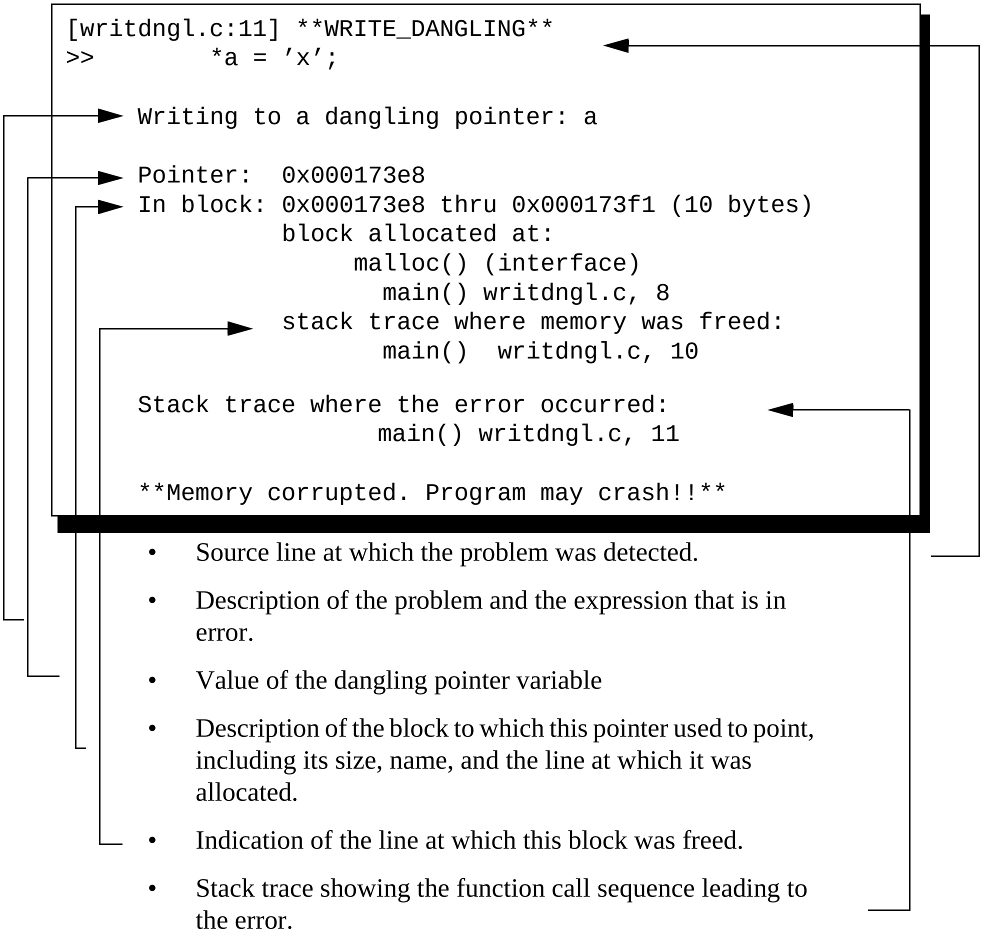
Problem

This code attempts to use a piece of dynamically allocated memory after it has already been freed.



```
/*
 * File: writdngl.c
 */
#include <stdlib.h>

main()
{
     char *a = (char *)malloc(10);

     free(a);
     *a = ’x’;
     return (0);
}
```

Diagnosis (at runtime)



```
[writdngl.c:11] **WRITE_DANGLING**
>>        *a = ’x’;

      Writing to a dangling pointer: a

      Pointer:  0x000173e8
      In block: 0x000173e8 thru 0x000173f1 (10 bytes)
                block allocated at:
                     malloc() (interface)
                       main() writdngl.c, 8
                stack trace where memory was freed:
                       main()  writdngl.c, 10

      Stack trace where the error occurred:
                       main() writdngl.c, 11

      **Memory corrupted. Program may crash!!**
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Value of the dangling pointer variable
- Description of the block to which this pointer used to point, including its size, name, and the line at which it was allocated.
- Indication of the line at which this block was freed.
- Stack trace showing the function call sequence leading to the error.

Repair

Check that the de-allocation that occurs at the indicated location should, indeed, have taken place. Also check that the pointer you are using should really be pointing to a block allocated at the indicated place.

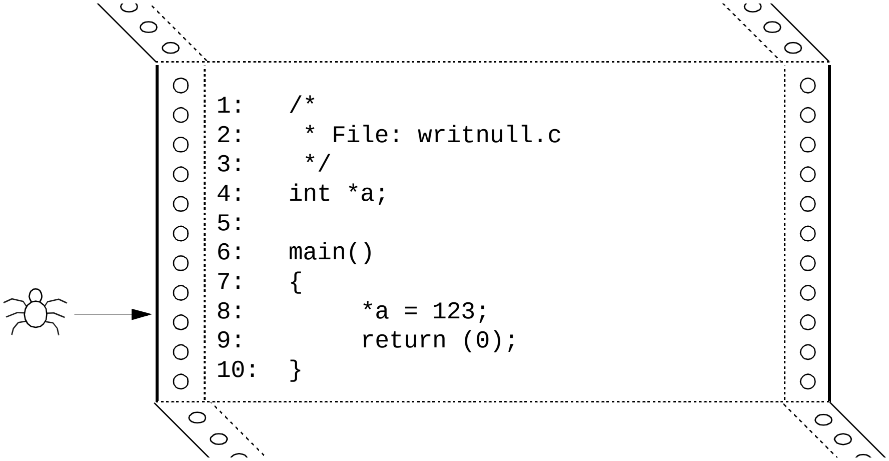
WRITE_NULL

Writing to a NULL pointer

This error is generated whenever an attempt is made to dereference a NULL pointer.

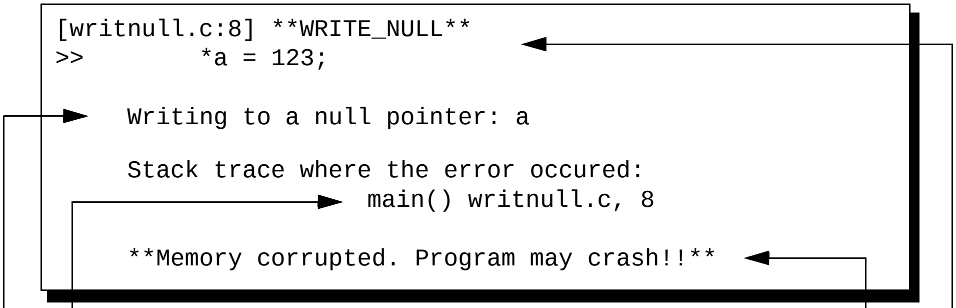
Problem

This code attempts to use a pointer which has not been explicitly assigned. Since the variable a is global, it is initialized to zero by default, which results in dereferencing a NULL pointer in line 8.



```
1:   /*
2:    * File: writnull.c
3:    */
4:   int *a;
5:
6:   main()
7:   {
8:        *a = 123;
9:        return (0);
10:  }
```

Diagnosis (at runtime)



```
[writnull.c:8] **WRITE_NULL**
>>        *a = 123;

     Writing to a null pointer: a

     Stack trace where the error occured:
                      main() writnull.c, 8

     **Memory corrupted. Program may crash!!**
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Stack trace showing the function call sequence leading to the error.
- Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

A common cause of this problem is the one shown in the example - use of a pointer that has not been explicitly assigned and which is initialized to zero. This is usually due to the omission of an assignment or allocation statement which would give the pointer a reasonable value.

The example code might, for example, be corrected as follows

```
/*
 * File: writnull.c (Modified)
 */
int *a;

main()
{
    int b;

    a = &b;
    *a = 123;
    return (0);
}
```

A second common source of this error is code which dynamically allocates memory but then zeroes pointers as blocks are freed. In this case, the error would indicate reuse of a freed block.

A final common problem is caused when one of the dynamic memory allocation routines, `malloc`, `calloc`, or `realloc`, fails and returns a `NULL` pointer. This can happen either because your program passes bad arguments or simply because it asks for too much memory. A simple way of finding this problem with Insure++ is to enable the `RETURN_FAILURE` error code (see page 342) via your `.psrc` file and run the program again. It will then issue diagnostic messages every time a system call fails, including the memory allocation routines.

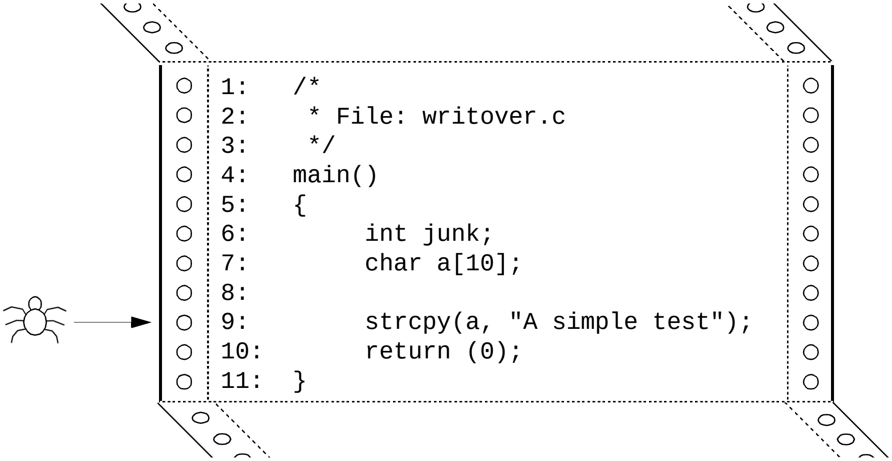
WRITE_OVERFLOW

Writing overflows memory

This error is generated whenever a block of memory indicated by a pointer will be written outside its valid range.

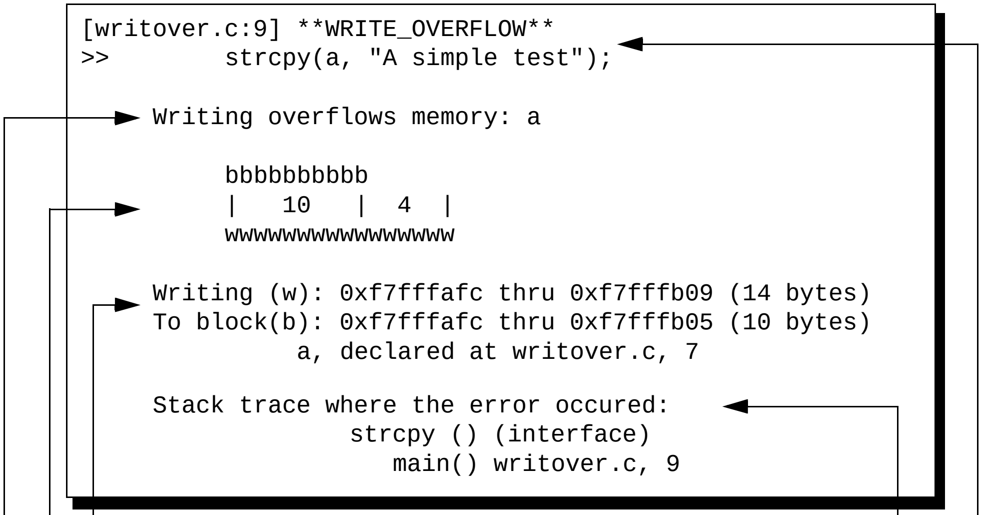
Problem

This code attempts to copy a string into the array a, which is not large enough.



```
/*
 * File: writover.c
 */
main()
{
     int junk;
     char a[10];

     strcpy(a, "A simple test");
     return (0);
}
```

Diagnosis (at runtime)



```
[writover.c:9] **WRITE_OVERFLOW**
>>         strcpy(a, "A simple test");

     Writing overflows memory: a

          bbbbbbbbbb
          |   10   |  4  |
          wwwwwwwwwwwwwwww

     Writing (w): 0xf7fffafc thru 0xf7fffb09 (14 bytes)
     To block(b): 0xf7fffafc thru 0xf7fffb05 (10 bytes)
                    a, declared at writover.c, 7

     Stack trace where the error occured:
                   strcpy () (interface)
                      main() writover.c, 9
```

- Source line at which the problem was detected.
- Description of the problem and the incorrect expression.
- Schematic showing the relative layout of the actual memory block (b) and region being written (w). (See “Overflow diagrams” on page 169.)
- Range of memory being written and description of the block to which write is taking place, including its size and the location of its declaration.
- Stack trace showing the call sequence leading to the error.

Repair

This error often occurs when working with strings.

In most cases, a simple fix is to increase the size of the destination object.

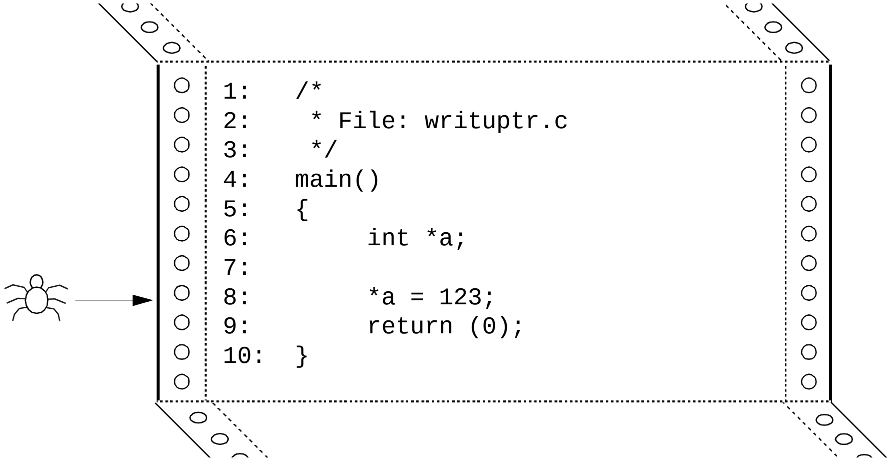
WRITE_UNINIT_PTR

Writing to an uninitialized pointer

This error is generated whenever an uninitialized pointer is dereferenced.

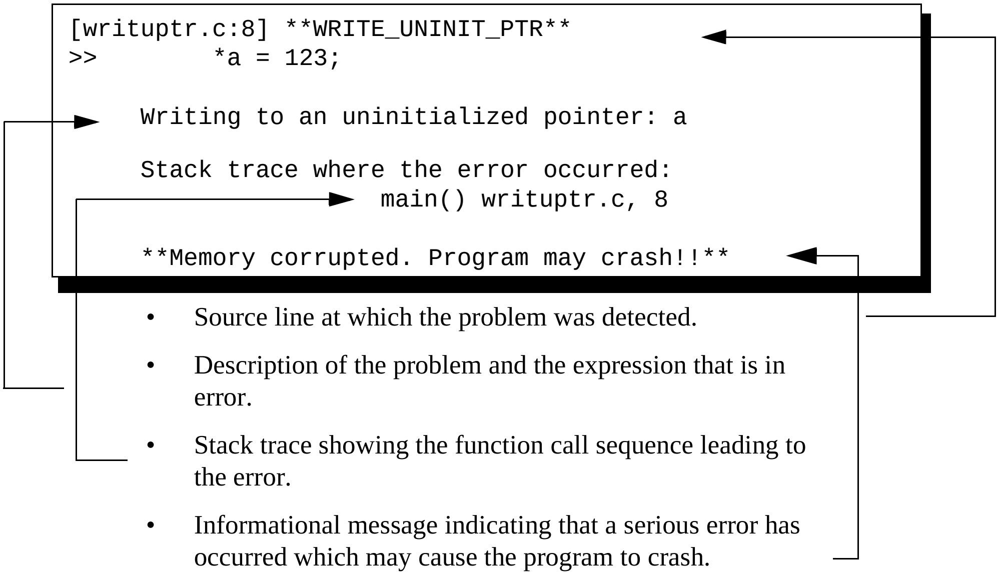
Problem

This code attempts to use the value of the pointer a, even though it has not been initialized.



```
/*
 * File: writuptr.c
 */
main()
{
        int *a;

        *a = 123;
        return (0);
}
```

Diagnosis (at runtime)



```
[writuptr.c:8] **WRITE_UNINIT_PTR**
>>        *a = 123;

     Writing to an uninitialized pointer: a

     Stack trace where the error occurred:
                      main() writuptr.c, 8

     **Memory corrupted. Program may crash!!**
```

- Source line at which the problem was detected.
- Description of the problem and the expression that is in error.
- Stack trace showing the function call sequence leading to the error.
- Informational message indicating that a serious error has occurred which may cause the program to crash.

Repair

This problem is usually caused by omitting an assignment or allocation statement that would initialize a pointer. The code given, for example, could be corrected by including an assignment as shown below.

```
/*
 * File: writuptr.c (Modified)
 */
main()
{
      int *a, b;

      a = &b;
      *a = 123;
}
```

WRITE_WILD

Writing to a wild pointer

This problem occurs when an attempt is made to dereference a pointer whose value is invalid or which Insure++ did not see allocated.

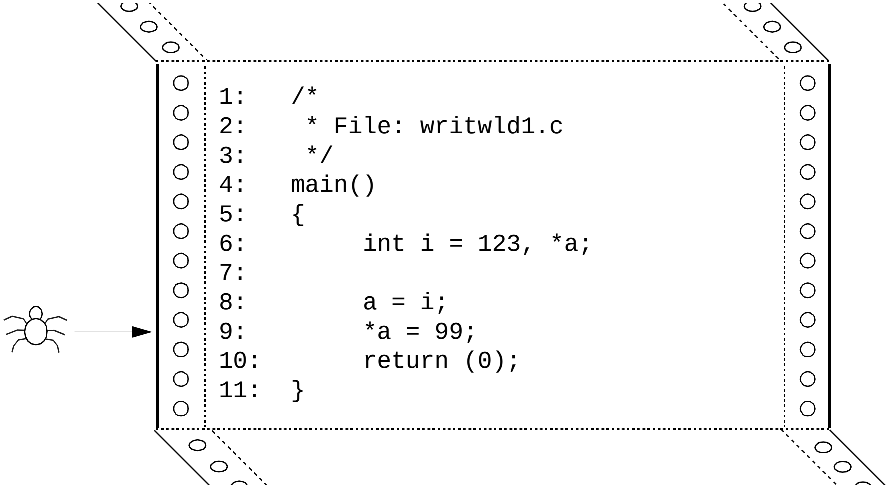
This can come about in several ways:

- Errors in user code that result in pointers that don't point at any known memory block.
- Compiling only *some* of the files that make up an application. This can result in Insure++ not knowing enough about memory usage to distinguish correct and erroneous behavior.

This discussion centers on the first type of problem described here. A detailed discussion of the second topic, including samples of its generation and repair can be found in "Interfaces" on page 103.

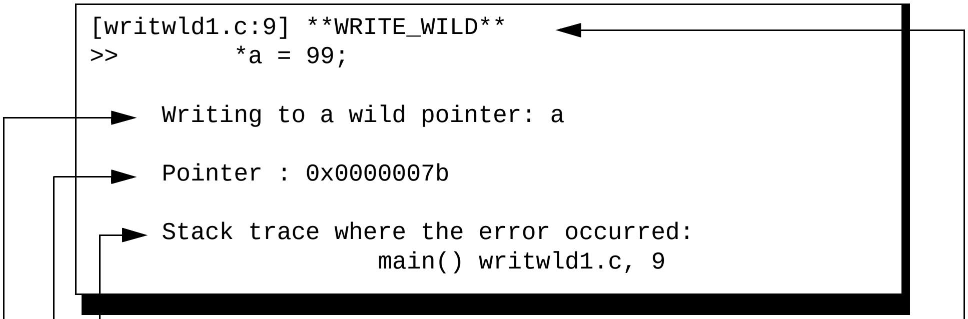
Problem #1

The following code attempts to use the address of a variable but contains an error at line 8 - the address operator (&) has been omitted.



```
1:   /*
2:    * File: writwld1.c
3:    */
4:   main()
5:   {
6:        int i = 123, *a;
7:
8:        a = i;
9:        *a = 99;
10:       return (0);
11:  }
```

Diagnosis (at runtime)



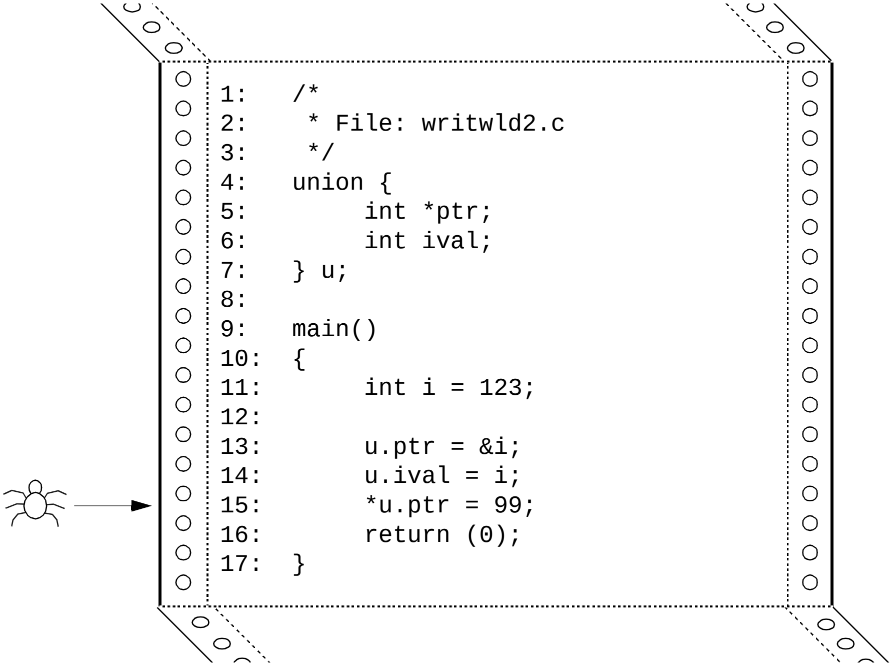
```
[writwld1.c:9] **WRITE_WILD**
>>        *a = 99;

     Writing to a wild pointer: a

     Pointer : 0x0000007b

     Stack trace where the error occurred:
                    main() writwld1.c, 9
```

- Source line at which the problem was detected.
- Description of the problem and the name of the parameter that is in error.
- Value of the bad pointer.
- Stack trace showing the function call sequence leading to the error.

Note that most compilers will generate warning messages for this error since the assignment in line 8 uses incompatible types.

Problem #2

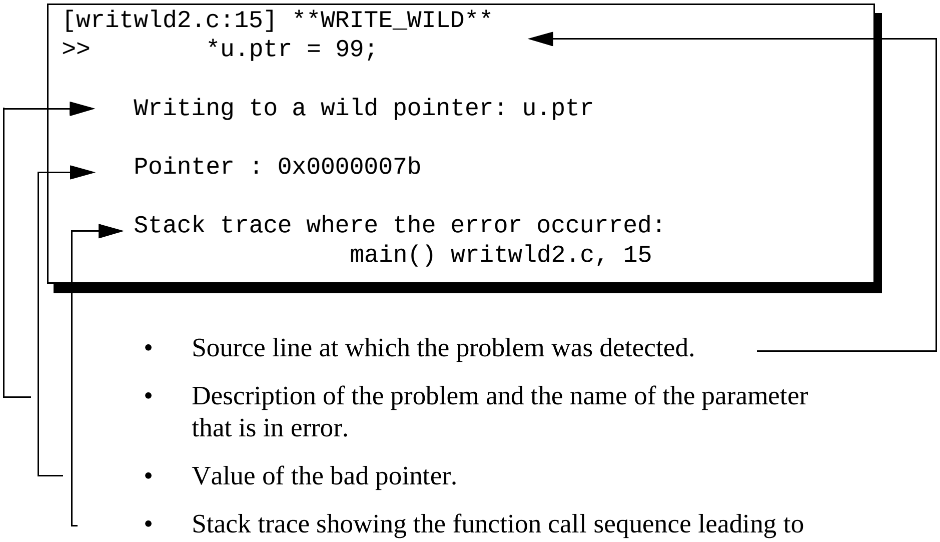
A more insidious version of the same problem can occur when using `union` types. The following code first assigns the pointer element of a union but then overwrites it with another element before using it.



```
/*
 * File: writwld2.c
 */
union {
     int *ptr;
     int ival;
} u;

main()
{
     int i = 123;

     u.ptr = &i;
     u.ival = i;
     *u.ptr = 99;
     return (0);
}
```

Note that this code will not generate compile time errors.

Diagnosis (at runtime)



```
[writwld2.c:15] **WRITE_WILD**
>>        *u.ptr = 99;

     Writing to a wild pointer: u.ptr

     Pointer : 0x0000007b

     Stack trace where the error occurred:
                    main() writwld2.c, 15
```

- Source line at which the problem was detected.
- Description of the problem and the name of the parameter that is in error.
- Value of the bad pointer.
- Stack trace showing the function call sequence leading to the error.

Repair

The simpler types of problem are most conveniently tracked in a debugger by stopping the program at the indicated source line. You should then examine the illegal value and attempt to see where it was generated. Alternatively you can stop the program at some point shortly before the error and single-step through the code leading up to the problem.

Note that wild pointers can also be generated when Insure++ has only partial information about your program’s structure. This issue is discussed extensively in “Interfaces” on page 103.

Insure++ API

This section lists the Insure++ functions that can be called from either an application program or a high level debugger.

If you are inserting these routines into your source code, you may want to use the pre-processor symbol `__Insure__` so that they will only be called when compiling with the appropriate tools, e.g.,

```
/*
 * Code being checked with Insure++
 */
      ...
/*
 * Disable runtime memory checking
 */
#ifdef __INSURE__
      _Insure_set_option("runtime", "off");
#endif
          ...
/*
 * Block of code without runtime checking...
 */
          ...
/*
 * Re-enable runtime checking
 */
#ifdef __INSURE__
      _Insure_set_option("runtime", "on");
#endif
```

In this way you can use the same source code when compiling with or without Insure++.

Control routines

These routines affect the behavior of Insure++ and are normally called from within your source code.

- `void _Insure_cleanup(void);`

  Causes the Insure++ runtime to close any open files properly.

- `void _Insure_printf(char *fmt,[,arg...]);`

  Causes Insure++ to add the given character string to its output.

- `void _Insure_set_option(char *option, char *value);`

  Used to override at runtime options which are set in `.psrc` files. The first argument is the option name and the second is the option argument that might be found in the `.psrc` file, e.g.

  ```
  _Insure_set_option("summarize", "bugs");
  ```

Memory block description routines

These functions can be called either from a program or from the debugger as described in “Interacting with Debuggers” on page 31.

- `long _Insure_list_allocated_memory(void);`

  Prints a list of all allocated memory blocks and their sizes. Returns the total number of bytes allocated.

- `void _Insure_mem_info(void *ptr);`

  Displays all information known about the memory block whose address is `ptr`. For example, the following code

```
#include <stdlib.h>

main()
{
      char *p, buf[128];

      p = malloc(100);
#ifdef __INSURE__
      _Insure_mem_info(buf);
      _Insure_mem_info(p);
#endif
           ...
```

might generate the following output

```
Pointer:   0xf7fff74c (stack)
Offset:    0 bytes
In block:  0xf7fff74c thru 0xf7fff7cb (128 bytes)
           buf, declared at foo.4, 4
Pointer:   0x00024b98 (heap)
Offset:    0 bytes
In block:  0x00024b98 thru 0x00024bfb (100 bytes)
           p, allocated at foo.c, 6
```

- `void _Insure_ptr_info(void **ptr);`

  Displays all information about the pointer whose address is passed. For example, the code

```
#include <stdlib.h>

main()
{
      char *p, buf[128];

      p = malloc(100);
#ifdef __INSURE__
      _Insure_ptr_info(&p);
#endif
          ...
```

might generate the following output

```
Pointer:  0x00024b98 (heap)
Offset:   0 bytes
In block: 0x00024b98 thru 0x00024bfb (100 bytes)
          p, allocated at foo.c, 6
```

Interface Functions

This section lists the Insure++ specific functions that can be called from Insure++ interface files. The use of these functions is described in the section “Interfaces” on page 103. This description gives only a brief summary of the purpose and arguments of the various functions. Probably the best way to see their purpose is to look at the source code for the interfaces shipped with Insure++, which can be found in subdirectories of the main Insure++ installation directory with names such as `src.solaris/cc`, `src.hp10/cc`, etc.

Note that these functions, despite appearances, are not C functions that you can insert into your C code. They can *only* be used in Insure++ interface files to be compiled with `iic`.

Memory Block Declaration Routines

These functions are used to indicate the usage of memory blocks. They do not actually allocate or free memory.

- `void iic_alloc(void *ptr, unsigned long size);`
  Declares a block of uninitialized heap memory of the given size. (See page 112.)
- `void iic_alloca(void *ptr, unsigned long size);`
  Declares a block of data on the stack.
- `void iic_alloci(void *ptr, unsigned long size);`
  Declares a block of initialized heap memory of the given size.(Without the second argument, declares a block the length of the first argument treated as a character string, including the terminating `NULL`.) (See page 116.)
- `void iic_allocs(void *ptr, unsigned long size);`
  Declares a pointer to a block of static memory. (Without the second argument, declares a block the length of the first

argument treated as a character string, including the terminating `NULL`.) (See page 116.)

-
  ```
  void iic_realloc(void *old, void *new,
          unsigned long new_size);
  ```
  Indicates that the indicated block has been re-allocated and that the contents of the old block should be copied to the new one.

-
  ```
  void iic_save(void *ptr);
  ```
  Specifies that the indicated block should never be reported to have “leaked”. Normally, this is used when the system will keep track of a memory block, even after all user pointers have gone.

-
  ```
  void iic_unalloc(void *ptr);
  ```
  Deallocates a block of memory. (See page 116.)

-
  ```
  void iic_unallocs(void *ptr);
  ```
  Undoes the effect of an `iic_allocs`. No error checking is performed on the pointer - the block is simply forgotten.

Memory Checking Routines

These functions report the appropriate Insure++ error message if the check fails.

-
  ```
  void iic_copy(void *to, void *from,
          unsigned long size);
  ```
  Checks for write and read access to the given number of bytes and indicates that the `from` block will be copied onto the `to` block. (See page 112.)

-
  ```
  void iic_copyattr(void *to, void *from)
  ```
  Copies the attribute properties (e.g., opaque, uninitialized, etc.) from one pointer to another.

-
  ```
  void iic_dest(void *ptr, unsigned long size);
  ```
  Checks for write access to the `ptr` for `size` number of bytes. (See page 116.)

-
  ```
  void iic_freeable(void *ptr);
  ```
  Checks that the indicated pointer indicates a dynamically allocated block of memory that could be freed.

-
  ```
  void iic_justcopy(void *to, void *from,
      unsigned long size);
  ```

Indicates that the `from` block will be copied onto the `to` block without performing the other `iic_copy` checks.

- `void iic_pointer(void *ptr);`

  Checks that the indicated pointer is valid, without checking anything about the size of the block it points to.

- `void iic_resize(void *ptr, unsigned long newsize);`

  Indicates that the block of memory has changed size.

- `void iic_source(void *ptr, unsigned long size);`

  Checks for read access to `ptr` for `size` bytes. (See page 116.) Does no checks for initialization of the block.

- `void iic_sourcei(void *ptr, unsigned long size);`

  Checks for read access to `ptr` for `size` bytes, and also that the memory is initialized. (See page 116.)

- `int iic_string(char *ptr, unsigned long size);`

  Checks that the pointer indicates a `NULL` terminated string. (See page 116.) If the optional second argument is supplied, the check terminates after at most that number of characters. In either case, the string length is returned, or -1 if some error prevented the string length from being computed.

Function Pointer Checks

- `void iic_declfunc(void (*f)());`

  Declares that the indicated pointer is a function regardless of appearance or other information.

- `void iic_func(void (*f)());`

  Checks that the indicated pointer is actually a function.

Opaque Types

- `void iic_opaque(void *ptr);`
  Declares that the pointer should never be checked.
- `void iic_opaque_type(<typename>);`
  Declares that the indicated structure or union type is opaque and should not be checked. (See page 72.)
- `void iic_opaque_subtype(<typename>, <typetag>);`
  Declares that the indicated element of a structure or union type is opaque and should not be checked. (See page 72.)

Printf/scanf Checking

- `void iic_input_format(char *format_string);`
  Indicates that the indicated string and the arguments following it should be checked as though they were a `scanf` style format string. This function should be called from the interface before activating the function being checked.
- `void iic_output_format(char *format_string);`
  Indicates that the indicated string and the arguments following it should be checked as though they were a `printf` style format string.
- `void iic_post_input_format(int ntokens);`
  This function can be called after an `iic_input_format` check and a call to an input function to check that the indicated number of tokens did not corrupt memory when read. If the argument is omitted, all the tokens from the `iic_input_format` string are checked.
- `int iic_strlenf(char *format_string, ...);`
  Returns the length of the string after substitution of the subsequent arguments, which are interpreted as a `printf` style format string.
- `int iic_vstrlenf(char *format_string, va_list ap);`
  Returns the length of the string after substitution of the argument, which must be the standard type for a variable

argument list. The `format_string` argument is interpreted in the normal `printf` style.

Utility Functions

- `char *iic_c_string(char *string);`

  Converts a string to a format consistent with the C language conventions. Useful for printing error messages.

-
  ```
  void iic_error(int code, char *format,
                  ...);
  ```

  Generates a message with the indicated error code (either `USER_ERROR` or `RETURN_FAILURE`). (See page 117.)

- `void iic_expand_subtype(<typename>, <typetag>);`

  Indicates that the structure or union named `typename` contains an element name `typetag` whose size varies at runtime. Normally used for "stretchy" arrays. (See page 72.) For example, if you have the following code, and `a` is a stretchy array,

  ```
  struct test {
      char a[1];
  };
  ```

  then the appropriate function call would be:

  ```
  iic_expand_subtype(struct test, a);
  ```

- `int iic_numargs(void);`

  Returns the number of arguments actually passed to the function.

- `void iic_warning(char *string);`

  Prints the indicated string at compile-time.

Callbacks

- `iic_body`

  Keyword used in function declarations to indicate that the function for which the interface is being specified will be used as a callback. (See “Using iic_body” on page 125.)

- `void iic_callback(void (*f)(),void (*template)());`

  Specifies that the function `f` will be used as a callback, and that whenever it is called its invocation is to be processed as indicated by the previously declared (static) function `template`. (See “Using iic_callback” on page 124.)

- `void iic_opaque_callback(void (*f)());`

  Specifies that the function `f` will be used as a callback, and that all of its arguments should be treated as opaque whenever it is invoked. (See “Which to use: iic_callback or iic_body?” on page 126.)

Variable Arguments

- `__dots__`

  Placeholder for variable arguments (“`...`”) in an argument list. (See page 121.)

Initialization

- `void iic_startup(void)`

  Function that can be declared in any interface file and which contains calls to be made before any function in the interface is executed. (See page 121.)

Termination

- `void iic_exit(void)`

  Indicates that the function specified by the interface is going to exit. This allows Insure++ to close its files and perform any necessary cleanup activity before the program terminates.

Manual Pages

The following pages contain UNIX-style manual pages for Insure++.

They are divided into two sections, as is conventional

- Commands that are invoked from the shell.
- System calls that are called from source code.

NAME

`iic` - Insure++ interface compiler

SYNOPSIS

```
iic   [-compiler name] [-Dsymbol[=value]]
      [-Idirectory] [-p] [-t] [-v] files
```

DESCRIPTION

This command is used to compile Insure++ interface files. Each source file is compiled into a similarly named file with the suffix `.tqs` that can be passed to the `insight` command. These files indicate the runtime behavior of routines whose source code was not processed by Insure++, and can also provide additional user level parameter checks.

OPTIONS

`-compiler name`

Indicates that the named compiler will be used to process the source code when `insight` is run, overriding the default or any value found in a `.psrc` file. This switch can affect the default directories searched for header files and pre-defined preprocessor symbols.

`-Dsymbol=value` Defines preprocessor symbols in the conventional C manner.

`-Idirectory` Add a directory to the path searched for header files.

`-p` If a prototype for a function exists (possibly in a header file), use its definition to override type mismatches in a function declaration.

`-t` Process the file as usual and generate (on `stdout`) a table summarizing the behavior of the routines defined.

`-v` Enable verbose mode. `iic` prints commands as it executes them.

EXAMPLES

```
iic mylib.c
```

Compiles the interface code in `mylib.c` and generates `mylib.tqs`.

```
iic -compiler gcc my_gnu_lib.c
```

Compiles the interface code in `my_gnu_lib.c` using the GNU C compiler `gcc`.

SEE ALSO

`iiinfo`, `iiwhich`, `insure`

NAME

`iiinfo` - Display information about an Insure++ interface file

SYNOPSIS

```
iiinfo [-e|m] [-s] [-v] tqsfile1 tqsfile2 …
```

DESCRIPTION

This command reads the specified Insure++ interface file and displays information about the contents. With no switches, `iiinfo` displays the names of the objects described in the interface file in the format

```
KeyLetter      Name
```

where `KeyLetter` describes the type of object being named and is one of

F	<Type>	Function (<Type> = linkable, inlined, or prototype)
T		Data type
V		Variable

OPTIONS

`-e`	Demangle C++ function names with extended attributes.
`-m`	Leave C++ function names mangled.
`-s`	Gives a summary of the contents of the named file. Indicates the amount of space given to each type of object.
`-v`	Adds the size of each object to its description.
`tqsfile1`	All other arguments name Insure++ interface files to be processed.

EXAMPLES

```
iiinfo foo.tqs
```

Displays the list of contents of the interface file `foo.tqs`.

SEE ALSO

`iic`, `iiwhich`

NAME

`iiusers` - Display all Insure++ licenses currently in use

SYNOPSIS

```
iiusers
```

DESCRIPTION

This command is used to display the Insure++ licenses that are currently in use. The information displayed includes the total number of licenses available, the total number of licenses in use, the userids of all users with licenses, and the time when each user's license will expire and become available for another user.

OPTIONS

none

EXAMPLES

```
iiusers
```

Display the current licenses in use. The output will look something like this:

```
Licenses In Use (1 of 1)
===============
insrel - expires in 15 minutes
```

This example shows that user insrel currently has the single available license, which will become available for another user in 15 minutes.

SEE ALSO

`psrcdump, pslic`

NAME

iiwhich - Search for and display an Insure++ interface description

SYNOPSIS

```
iiwhich [-compiler <compiler_name>] [-l]
      [-v] [library_files] funcname
```

DESCRIPTION

This command searches a set of Insure++ interface files for definitions of the named routines. For each routine, the corresponding definition is displayed. If the interface is a linkable one, iiwhich will not be able to display the actions taken by the interface, but it will indicate the prototype for the function.

If any of the names on the command line have the suffix ".tqs" or ".tqi", they are taken to be Insure++ interface modules and added to the list from which to search.

OPTIONS

-compiler <compiler_name>
Use compiler <compiler_name> for interface lookup.

-l
Displays the list of .tqs/.tqi files in the order that they will be processed. Useful for resolving conflicts between multiple interface specifications.

-v
Displays each .psrc file name as it is traversed, preceded by a # character.

library_files
Any files whose names end in the suffix ".tqs" or ".tqi" are taken to be Insure++ interface files and are added to the default list of interface modules in which names will be searched.

`funcname`

All other arguments are treated as function names and are searched for in the standard interface modules (and any others given as command line arguments or in `.psrc` files). If a matching definition is found, its source file and definition are displayed.

EXAMPLES

```
        iiwhich malloc
```

Searches for the definition of the C library function `malloc`.

```
        iiwhich mylib.tqs myfunc1 myfunc2
```

Searches for definitions of functions `myfunc1` and `myfunc2` in both the standard library interface modules and also the interface file `mylib.tqs`.

SEE ALSO

`iic`, `insure`, `psrcdump`

NAME

imangle - Display Insure++'s mangled C++ function names for a given file

SYNOPSIS

```
imangle function_name filename
```

DESCRIPTION

This command searches the given file (and any headers and #includes) for functions matching function_name. For each match, the Insure++ mangled name will be displayed along with a filename and line number.

The mangled name is often useful to get Insure++ to distinguish between different versions of an overloaded function.

This can be used in conjunction with the function_ignore .psrc option (see the "Configuration Files" section of the *User's Guide* for more details about this option) to do no checking on certain functions.

EXAMPLE

```
imangle func file.C
```

Displays the mangled name of function func in file file.C.

SEE ALSO

insure

NAME

`ins_ld` - Link programs with Insure++

SYNOPSIS

```
ins_ld [-Zoi “option”]
      [-Zop option_file] [-Zsl] [-Zvm]
      [library_files] <linker_arguments>
```

DESCRIPTION

This command takes the place of your normal link command if you are using Insure++ (typically the linker is called `ld`). This should only be used if you are linking your program explicitly. If you are using the compiler command to link your program, you should use the `insure` command.

OPTIONS

`-Zoi “option”` Treat `option` as a `.psrc` option. Multiple `-Zop` files and `-Zoi` options will be processed in order from left to right before any source files are processed.

`-Zop option_file`

Process the named file as though it were an additional `.psrc` file. This allows options to be supplied on the command line that override those of the other `.psrc` files. Multiple `-Zop` files and `-Zoi` options will be processed in order from left to right before any source files are processed.

`-Zsl` Perform a “safe link”. Normally, Insure++ forces every object file and library to be linked into the executable, without exception. This can occasionally cause conflicts if symbols are defined multiple times in different libraries. This option performs a slower link that avoids such problems by only linking files that are actually required. This option is tried automatically if fault recovery is turned on (the default).

`-Zvm` Enable verbose mode. Displays each command as it is executed.

`library_files` Any files whose names end in the suffix `.tqs, .tqi, or .tql` are taken to be Insure++ interface files and are processed in conjunction with the source files on the command line.

`<linker_arguments>`
All other arguments are passed directly to your normal linker (or the one specified in a `.psrc` file using a `linker` directive).

EXAMPLES

```
insure -c hello.c
ins_ld -e start -o hello \
    /usr/lib/crt0.o hello.o -lc
```

Compile and link the C source file `hello.c`.

SEE ALSO

`insure`, `.psrc` (lists configuration options)

NAME

`insure` - Compile and link programs with Insure++

SYNOPSIS

```
insure [-Zlh] [-Zoi “option”]
      [-Zop option_file] [-Zsl] [-Zvm]
      [library_files] <compiler_arguments>
```

DESCRIPTION

This command takes the place of your normal compiler. In addition to compiling and linking your program, `insure` will insert code to monitor memory accesses and check for runtime bugs.

If errors can be detected at compile time, an appropriate message will be printed. Otherwise, all checks are done at runtime.

The pre-processor symbol `__INSURE__` is automatically defined whenever you are using `insure`, so you may use this to conditionally include or exclude program fragments.

OPTIONS

`-Zlh`
Specify where on the command line the additional Insure++ libraries should be placed - the `-Zlh` flag (“link here”) is replaced with the names of the Insure++ libraries. This is only necessary if the default location is incorrect.

`-Zoi “option”`
Treat `option` as a `.psrc` option. Multiple `-Zop` files and `-Zoi` options will be processed in order from left to right before any source files are processed.

`-Zop option_file`
Process the named file as though it were an additional `.psrc` file. This allows options to be supplied on the command line that override those of the other `.psrc` files. Multiple `-Zop` files and `-Zoi` options will be processed in order from left to right before any source files are processed.

`-Zsl` Perform a “safe link”. Normally, Insure++ forces every object file and library to be linked into the executable, without exception. This can occasionally cause conflicts if symbols are defined multiple times in different libraries. This option performs a slower link that avoids such problems by only linking files that are actually required. This option is tried automatically if fault recovery is turned on (the default).

`-Zvm` Enable verbose mode. Displays each command as it is executed.

`library_files` Any files whose names end in the suffix `.tqs, .tqi,` or `.tql` are taken to be Insure++ interface files and are processed in conjunction with the source files on the command line.

`<compiler_arguments>`

All other arguments are passed directly to your normal compiler (or the one specified in a `.psrc` file using a `compiler` directive).

EXAMPLES

```
insure -o hello hello.c
```

Compile and link the C source file `hello.c`.

```
insure mylib.tqs -c file1.c
insure mylib.tqs -c file2.c
insure -o myprog file1.o file2.o
```

These commands compile the files `file1.c` and `file2.c`, including checks from the Insure++ interface file `mylib.tqs`. The resulting object files are linked together into a program called `myprog`.

```
insure -Zoi “compiler gcc” -o foo foo.c
insure -o foo foo.c -Zoi “compiler gcc”
```

These commands are completely equivalent and build the program `foo` with the compiler `gcc`.

SEE ALSO

`iic`, `ins_ld`

NAME

`pslic` - ParaSoft license manager

SYNOPSIS

```
pslic [-hostid]
```

DESCRIPTION

This command is used to add or delete licenses for all ParaSoft tools. You will need to have write permission for the `.psrc` file in the main ParaSoft installation directory before running this program. If you need to call ParaSoft for a password, you should be prepared to run this command on the machine which will be running the software.

OPTIONS

`-hostid` prints current machine/network ID.

EXAMPLES

```
      pslic
```

Modify the current licenses. The output will look something like this:

```
ParaSoft License Manager Version 1.1
Copyright (C) 1998-2000 by ParaSoft Corporation

This program enables you to examine or alter the
licenses for your ParaSoft tools.

Machine id: SOL-72769549
Network id: SOL-80c03756

You have the following tools installed:

 CodeWizard 2.0
 Insure++ 5.0
```

```
Inuse 5.2
TCA 5.2
Threads++ 5.2
Insure++ Package 5.2

Current licenses
================
      1: CodeWizard 3.2, Network SUN4-0x80000000,
                            expires Mar 29, 2000

Options:
      (A)dd a license
      (D)elete a license
      (M)odify a license
      (S)how machine and network id
      (E)xit and save changes
      (Q)uit without saving changes

Choose one:
```

Option (S) displays the ParaSoft host ids for the current machine and network. The output should look something like the following:

```
      Machine id: SUN4-23003555
      Network id: SUN4-80c03756
```

SEE ALSO

psrcdump

NAME

`psrcdump` - Search for and display all currently active `.psrc` options

SYNOPSIS

```
psrcdump [-a] [-compiler <compiler_name>]
      [-t tool] [-v]
```

DESCRIPTION

This command can be used to print out all `.psrc` options that are active in the current directory when running a ParaSoft tool, e.g. Insure++ or CodeWizard.

OPTIONS

`-a` Displays all options from the various `.psrc` files.

`-compiler <compiler_name>`
Specifies which compiler-default `.psrc` options you would like to display. If this option is not used, the options for the compiler specified in the `.psrc` files will be used.

`-t tool` Specifies which tool's options you would like to display.

`-v` Displays each `.psrc` file name and path as it is traversed, preceded by a '#' character.

EXAMPLES

```
psrcdump -t codewizard
```

Searches all `.psrc` files and prints all options which apply to CodeWizard.

SEE ALSO

`insure`

NAME

iic_body, iic_callback, iic_opaque_callback
- Insure++ interface routines to specify callback functions.

SYNOPSIS

```
<type> iic_body <name>(<arguments>) { ... }

void iic_callback(void (*f)(),
                  void (*template)());

void iic_opaque_callback(void (*f)());
```

WARNING

These functions may only be called from Insure++ interface modules which are compiled with the special iic compiler. They may not be inserted into regular code.

DESCRIPTION

These functions provide two ways of telling Insure++ that a function will be used as a callback - being invoked from within some code which is not compiled with Insure++. Typical examples include code invoked from utility functions such as qsort and scandir, signal handlers, and callback functions in the X Window System and other graphical user interfaces.

It is important to note that Insure++ does not require that you specify the behavior of callback functions. These routines are provided only to enhance the error checking performed at callback invocations by adding user-level checks.

The two methods described on this manual page are mutually exclusive for each callback - you should either use the iic_body method or the iic_callback method, but not both.

The iic_body keyword may be inserted in the definition of any function to indicate that it will be used in a callback role. Note that you *must* provide interface specifications with the iic_body specifier, even if the source code for the indicated function will be compiled with

Insure++, and that you must provide interfaces for every function that will be used as a callback.

`iic_callback` is used when specifying interfaces to functions which install callbacks. It connects the user-supplied callback function pointer with a “template” which is a statically declared interface function that indicates the type and number of arguments expected by the callback, and also any appropriate error checking that should be performed on these arguments before invoking the callback.

In the simple (and common) case, in which you wish to simply make the arguments to the called function opaque before invoking the callback, the simpler interface `iic_opaque_callback` is provided.

In this case, you need to specify the callback behavior *only* for the functions which register the callbacks - you need take no action for the functions which will actually be invoked as callback.

EXAMPLES

Assume that the function `myfunc1` is to be used as a callback function. It has two arguments, both strings, and returns an integer result.

A suitable callback interface would be

```
int iic_body myfunc1(char *p1, char *p2)
{
      iic_opaque(p1);
      iic_opaque(p2);
      return myfunc1(p1, p2);
}
```

Note that the interface makes both pointers opaque before invoking the callback. This prevents Insure++ from generating spurious messages in the case that the runtime system passes pointers to the callback which are not known to Insure++.

Note that this method requires that you make a separate interface for every function in your application which will be used as a callback.

The alternative to this approach is to make an interface to the routine which registers the callback, using the iic_callback routine. Assuming that the function install_callback is responsible for this process we could, instead, make the following interfaces.

```
static int callback_template(char *p1, char *p2)
{
      iic_opaque(p1);
      iic_opaque(p2);
      return callback_template(p1, p2);
}

int install_callback(void (*func)())
{
      iic_func(func);
      iic_callback(func, callback_template);
      return install_callback(func);
}
```

This interface first declares a (static) template which shows how the callback function will be invoked and also indicates the checks and/or actions that should be performed on its arguments before its invocation. It then uses the iic_callback function to connect the user-supplied function pointer argument and the callback template.

Since the template merely renders the arguments opaque before making the callback call, this interface could be re-coded more simply as

```
int install_callback(void (*func)())
{
      iic_func(func);
      iic_opaque_callback(func);
      return install_callback(func);
}
```

This approach requires that you make interfaces only for those functions which register callback procedures.

WARNING

The `iic_callback` function can only be used when the connection between the function pointer and the callback template will be used "immediately" and then dropped. For an explanation of what constitutes immediate use, consult the section "Which to use: iic_callback or iic_body?" on page 126 of the Insure++ *User's Guide*.

The `iic_opaque_callback` routine and `iic_body` keywords do not suffer from the same problems and can always be used.

SEE ALSO

`iic`

NAME

`iic_exit` - Specify that a Insure++ interface routine will not return.

SYNOPSIS

```
void iic_exit(void);
```

WARNING

This function may only be called from Insure++ interface modules which are compiled with the special `iic` compiler. It may not be inserted into regular code.

DESCRIPTION

Calling this function from an Insure++ interface function description indicates to the system that the function will terminate. This allows Insure++ to take any necessary precautions, such as closing open files and making summaries.

EXAMPLES

Execute the command `iiwhich exit` to see an example of the use of this function.

SEE ALSO

`iic`

NAME

`iic_declfunc`, `iic_func` - Insure++ interface routines to manipulate function pointers

SYNOPSIS

```
void iic_declfunc(void (*func)());
```

```
void iic_func(void (*func)());
```

WARNING

These functions may only be called from Insure++ interface modules which are compiled with the special `iic` compiler. They may not be inserted into regular code.

DESCRIPTION

`iic_declfunc` is used to tell Insure++ that the argument is a pointer to a function.

`iic_func` is used in interface modules to check that the supplied argument is a pointer to a function.

EXAMPLES

Both of these functions are used in the interface to the UNIX `signal` function. Initially, `iic_func` is used to check that the user-supplied argument is a function pointer. The value returned by `signal` is then declared to be a function pointer with the `iic_declfunc` routine.

This is done so that when the user overrides the system's default signal handler with a custom one, the value returned will henceforth be recognized to be a function pointer by Insure++, even if its source code was not compiled with the `insure` command.

SEE ALSO

`iic`

NAME

iic_copy, iic_copyattr, iic_dest, iic_freeable, iic_justcopy, iic_source, iic_sourcei, iic_pointer, iic_resize, iic_string - Insure++ interface routines to check data blocks

SYNOPSIS

```
void iic_copy(void *to, void *from,
                  unsigned long nbytes);

void iic_copyattr(void *to, void *from);

void iic_dest(void *to,
                  unsigned long nbytes);

void iic_freeable(void *ptr);

void iic_justcopy(void *to, void *from,
                  unsigned long nbytes);

void iic_source(void *from,
                  unsigned long nbytes);

void iic_sourcei(void *from,
                  unsigned long nbytes);

void iic_pointer(void *ptr);

void iic_resize(void *ptr,
                  unsigned long newsize);

int iic_string(char *str);

int iic_string(char *str, int nbytes);
```

WARNING

These functions may only be called from Insure++ interface modules which are compiled with the special `iic` compiler. They may not be inserted into regular code.

DESCRIPTION

These routines are used to check that memory blocks are large enough to perform various operations.

`iic_source` and `iic_dest` simply check that the indicated block of memory is large enough to read or write the indicated number of bytes, respectively. `iic_sourcei` is equivalent to `iic_source`, but performs an additional check that the block of memory is completely initialized.

`iic_copy` essentially combines the actions of `iic_source`, `iic_dest`, and `iic_justcopy` to move data from one buffer to another, checking both.

`iic_copyattr` copies the attribute properties (e.g., opaque, uninitialized, out of range) from one pointer to another. It is used to ensure that a return value from a function has the same properties as another pointer. For an example, see the interface to `strcpy`.

`iic_resize` is used to indicate to Insure++ that the memory block indicated by the first argument has changed size.

The `iic_string` routine exists in two forms. With only one argument, it checks to see that the supplied pointer points to a valid, NULL terminated string. With a second argument, it checks at most that number of characters before giving up. In either case, the number of characters in the string is returned, or -1 if some error prevents the string from being checked successfully.

`iic_pointer` simply checks that the argument is a valid pointer.

`iic_freeable` checks that the pointer argument indicates a block of dynamically allocated memory that could be freed. No other properties of the block are checked.

EXAMPLES

These functions form the basis for most of the library checking performed by Insure++ and can be found in many of the interface modules.

The use of the `iic_string` functions is most clearly demonstrated in the interfaces for functions such as `strcpy`, `strncpy` and `strlen`.

SEE ALSO

`iic`

NAME

iic_alloc, iic_alloca, iic_alloci, iic_allocs,
iic_realloc, iic_save, iic_unalloc,
iic_unallocs
- Insure++ interface routines to allocate and free memory blocks

SYNOPSIS

```
void iic_alloc(void *ptr,
                  unsigned long size);

void iic_alloca(void *ptr,
                  unsigned long size);

void iic_alloci(void *ptr,
                  unsigned long size);

void iic_allocs(void *ptr,
                  unsigned long size);

void iic_realloc(void *old, void *new,
                  unsigned long size);

void iic_save(void *ptr);

void iic_unalloc(void *ptr);

void iic_unallocs(void *ptr);
```

WARNING

These functions may only be called from Insure++ interface modules which are compiled with the special iic compiler. They may not be inserted into regular code.

DESCRIPTION

These routines are used to indicate that routines allocate and/or free areas of memory. The first four routines each indicate that their first argument is a pointer to size bytes of memory of various types, as follows:

iic_alloc	Uninitialized heap memory, such as that returned from the standard malloc routine.
iic_alloca	Stack memory, such as that allocated with alloca.
iic_alloci	Initialized heap memory, such as that obtained from the calloc function, or which has been initialized by the called routine itself.
iic_allocs	Static memory. The pointer points to a region of statically allocated (probably global) memory.

iic_realloc is used to indicate that a block has changed size and (possibly) location.

iic_save is used to prevent Insure++ from diagnosing memory leaks on a block of memory. In certain situations (for example, the callbacks to functions in windowing systems) a pointer is passed to a library routine which memorizes it internally. This is indicated with the iic_save call.

iic_unalloc indicates that the associated block has been freed.

iic_unallocs undoes the effect of a call to iic_allocs. No checking is performed on the indicated block - it simply disappears.

EXAMPLES

The interfaces to standard routines such as malloc, calloc, and free use these routines.

SEE ALSO

iic

NAME

`iic_opaque`, `iic_opaque_type`, `iic_opaque_subtype` - Insure++ interface routines to inhibit checking on data types

SYNOPSIS

```
void iic_opaque(void *ptr);

void iic_opaque_type(<typename>);

void iic_opaque_subtype(<typename>,
                  <typetag>);
```

WARNING

These functions may only be called from Insure++ interface modules which are compiled with the special `iic` compiler. They may not be inserted into regular code.

DESCRIPTION

`iic_opaque` can be used in any interface function to indicate that the pointer should never be checked for errors.

`iic_opaque_type` and `iic_opaque_subtype` may be called from the `iic_startup` routine in an interface file to indicate that the indicated structure type or subtype should never be checked.

SEE ALSO

`iic`

NAME

iic_input_format, iic_output_format, iic_strlenf, iic_vstrlenf - Insure++ interface routines to handle formatted I/O

SYNOPSIS

```
void iic_input_format(char *format);

void iic_output_format(char *format);

void iic_post_input_format(int ntokens);

int iic_strlenf(char *format, ...);

int iic_vstrlenf(char *format, va_list v);
```

WARNING

These functions may only be called from Insure++ interface modules which are compiled with the special iic compiler. They may not be inserted into regular code.

DESCRIPTION

iic_input_format checks that the indicated string and the arguments following it can be used as in a call to scanf. It should be called before invoking the actual routine being checked to determine if the argument types are valid. After calling the function, the routine iic_post_input_format can be called to check that none of the arguments corrupted memory. Only ntokens arguments will be checked, unless no argument is passed to iic_post_input_format, in which case all arguments are checked

iic_output_format checks that the indicated string and the arguments following it can be used as in a call to printf.

iic_strlenf and iic_vstrlenf return the length of a string after substitution of arguments according to the `printf` conventions.

EXAMPLES

`iic_input_format` is used in the interface to `scanf`.

`iic_output_format` is used in the interface to `printf`.

`iic_strlenf` is used in the interface to `sprintf`.

`iic_vstrlenf` is used in the interface to `vsprintf`.

SEE ALSO

`iic`

NAME

`iic_startup` - Initialization module for Insure++ interfaces

SYNOPSIS

```
void iic_startup(void);
```

DESCRIPTION

A function with this name may be placed in any Insure++ interface file. It contains calls which initialize important properties of the functions and data structures in that module. Any calls in this function will be executed prior to any interface code.

EXAMPLES

Executing the command

```
iiwhich iic_startup
```

lists all the interface files which contain `iic_startup` routines, and their contents.

SEE ALSO

`iic`

NAME

iic_c_string, iic_error, iic_expand_subtype, iic_numargs, iic_warning - Insure++ interface utility routines

SYNOPSIS

```
char *iic_c_string(char *string);

void iic_error(int code, char *fmt, ...);

void iic_expand_subtype(<typename>,
                  <typetag>);

void iic_numargs(void);

void iic_warning(char *string);
```

WARNING

These functions may only be called from Insure++ interface modules which are compiled with the special iic compiler. They may not be inserted into regular code.

DESCRIPTION

iic_c_string converts a string into a format consistent with the C language quoting conventions. It is useful for formatting error messages.

iic_error generates a standard format Insure++ error message with the indicated error code (either USER_ERROR or RETURN_FAILURE).

iic_expand_subtype is used to implement “stretchy arrays”. The two arguments specify a structure tag and subtype and tell Insure++ that the indicated structure element can change at runtime.

iic_numargs returns the number of arguments with which a function has been called.

Calling iic_warning from an interface causes the indicated string to be displayed every time the function is called. This string appears during compilation with the interface, not at runtime. It can be used to print messages informing the user that some feature is not fully checked.

EXAMPLES

`iic_c_string` and `iic_error` are used in many of the supplied interface functions. For one example, see the interface to `fopen`.

`iic_numargs` is used in the interface to `scanf`.

`iic_warning` is used in the interface to the UNIX `ioctl` function to indicate that the second and third arguments (which have widely varying data types) will not be checked.

SEE ALSO

`iic`

NAME

`__dots__` - Placeholder for a variable argument list in interface files.

SYNOPSIS

```
__dots__
```

WARNING

This value may only be used in Insure++ interface modules which are compiled with the special `iic` compiler. It may not be inserted into regular code.

DESCRIPTION

This pseudo-variable is used to indicate where in a function argument list the variable arguments indicated by the “`...`” notation should be inserted.

EXAMPLES

Execute the command `iiwhich printf` to see an example of the use of this function.

SEE ALSO

`iic`

Index